Analyzing NMAP

oprix

OPRIX 유성철

oprix@hanmail.net

들어가기 전에

한국의 네트워크는 많은 발전을 했다. 초창기 느린 모뎀 통신을 했던 시절도 있었지만, 지금에 와서는 네트워크 기반은 세계 최고 일 정도로 많은 발전을 했다. 그렇지만 Code Red 경우에서 보듯이, 네트워크의 양적인 성장을 했지만, 질 적인 성장은 하지 못했다. 네트워크의 문제점을 검사하는 도구도 많이 사용하지 않는 실정이다.

그래서 많은 도구 중에서 쉽게 접할 수 있고, 소스가 공개되어 있는 nmap이란 툴을 골랐다. 흔히 해킹 툴로 알려져 있지만, 그건 사용하기 나름이다. 날이 선 칼을, 음식 만들 때 사용하면 좋지만, 사람을 다치게 하는 나쁜 도구로 쓰이는 건 쓰는 사람에 달려있다.

이 문서는 네트워크의 보안상의 문제점을 분석하는 nmap 이란 툴의 소스를 분석하고, 그 소스의 내용을 분석하면서 나오는 여러 네트워크 스캔 방법을 얘기한다. 많은 관리자들이 nmap을 알고 있지만, nmap 의 깊은 곳까지 알지 못하고, 간단한 스캔에만 사용한다. 그래서 여러 스캔들의 이론에도 무게를 두었다. 단지 명령을 치고 결과를 기다리기 보다는 스캔에 대한 원리를 알고, 스캔을 해보는 게 좋겠다는 생각이다.

코드에 관한 깊은 이해와 네트워크 지식이 부족하기 때문에 세심하게 분석은 하지 않았다. 그렇지만, 이론적인 분석과 어느 부분을 분석하면 되는지에 대한 방향 설정을 해 두었다. 분석에 사용한 nmap 버전은 nmap-2.54BETA30 이었고, 현재 이 글을 쓰는 때는 nmap-2.54BETA34 버전이 나왔다.

차례

1 nmap 소개

nmap
nmap의 기본 동작 원리
nmap이 지원하는 스캔

2 open scan

open scan method
connect scan
connect scan simple code
reverse ident scan
reverse ident scan simple code

3 half open scan

half open scan method
syn scan
syn scan simple code
idle scan
idle scan simple code

4 stealth scan

stealth scan method
maimon scan
maimon scan simple code
fin scan
fin scan simple code
ack scan
ack scan simple code
window scan
window scan simple code
null scan
null scan simple code
xmas scan
xmas scan simple code

tcp fragmenting
tcp fragmenting simple code

5 sweep
icmp ping
icmp ping simple code
tcp echo
tcp echo simple code
tcp syn
tcp syn simple code
tcp ack
tcp ack simple code

6 miscellaneous
udp scan
udp scan simple code
ftp server bounce scan
ftp server bounce scan simple code

7 finger print
finger print 방법
finger print simple code

8 맺음말

9 참고 문헌

1
nmap 소개

nmap

nmap 은 시스템의 어떠한 포트가 열려 있나 보는데 사용하는 네트워크 스캐너이다. phrack 51-11 The Art of Scanning 에서 fyodor가 최초로 소개 하였고, GPL로 사용되게 계속 발전하고 있으며 Unix 계열에서 사용하는 가장 인기있는 스캐너 이다.

현재 command line 에서 사용하는 nmap , X-windows 상에서 사용하는 nmapFE 가 있으며, Windows에서 사용할 수 있지만 Unix 에서 사용할때 성능이 좋다. 또한 공개 취약점 스캐너인 Nessus 의 내부에서 사용하는 interactive 모드도 내장하고 있다.

nmap 의 기본 동작 원리

nmap 은 positive scan(pos_scan) 과 negative scan(super_scan) 의 두가지의 기본 동작 원리로 작동한다.

positive scan 은 시스템에 특정 패킷을 보내면 특정 timeout 내에 시스템의 열린 포트에서만 패킷에 반응을 한다. 이에 비해 negative scan 은 시스템에 특정 패킷을 보내면 닫힌 포트에서만 패킷에 반응한다.

positive scan 은 syn scan , ping scan , connect scan 이 있고,
negative scan 으로 fin scan , null scan , xmas scan 등이 있다.

nmap이 지원하는 scan

open

connect scan : connect 함수의 에러 값을 검사해서 포트가 닫혔는지 열렸는지 검사한다. 패킷을 변경하지 않으므로 root가 아닌 일반 사용자도 사용할 수 있다.

ident scan: identd 데몬 서비스를 이용해서 작동되는 포트와 연결된 데몬의 권한을 알 수 있다.

half open

SYN scan: SYN 패킷을 보내면 Three way handshaking 때문에 상대편에서 오는 SYN 패킷의 내용을 보고 포트가 닫혔는지 열렸는지 검사한다.

stealth

FIN scan: FIN 패킷을 보내면 포트가 닫혀 있을 경우, RST 패킷을 보내는 Unix 서버들의 특징을 이용해서 포트가 닫혔는지 열렸는지 검사한다.

Null scan: 어떤 flag 도 없는 패킷을 보내어 되돌아 오는 RST 값을 검사한다.

Xmas(all) scan: 모든 flag 를 세팅한 패킷을 보내어 RST 값을 검사한다.

ACK scan: 포트에 ACK 패킷을 보내어 RST 응답을 받으면 그 포트는 "unfilterd"이며, 아무런 응답이 없으면 "filtered" 이다.

Fragment scan: 패킷을 2개로 쪼개어 보낸다.

Window scan : ACK 패킷을 보내어 TCP Window 크기의 변화를 살핀다.

sweeps

Ping scan: icmp 의 echo 기능을 이용한다.

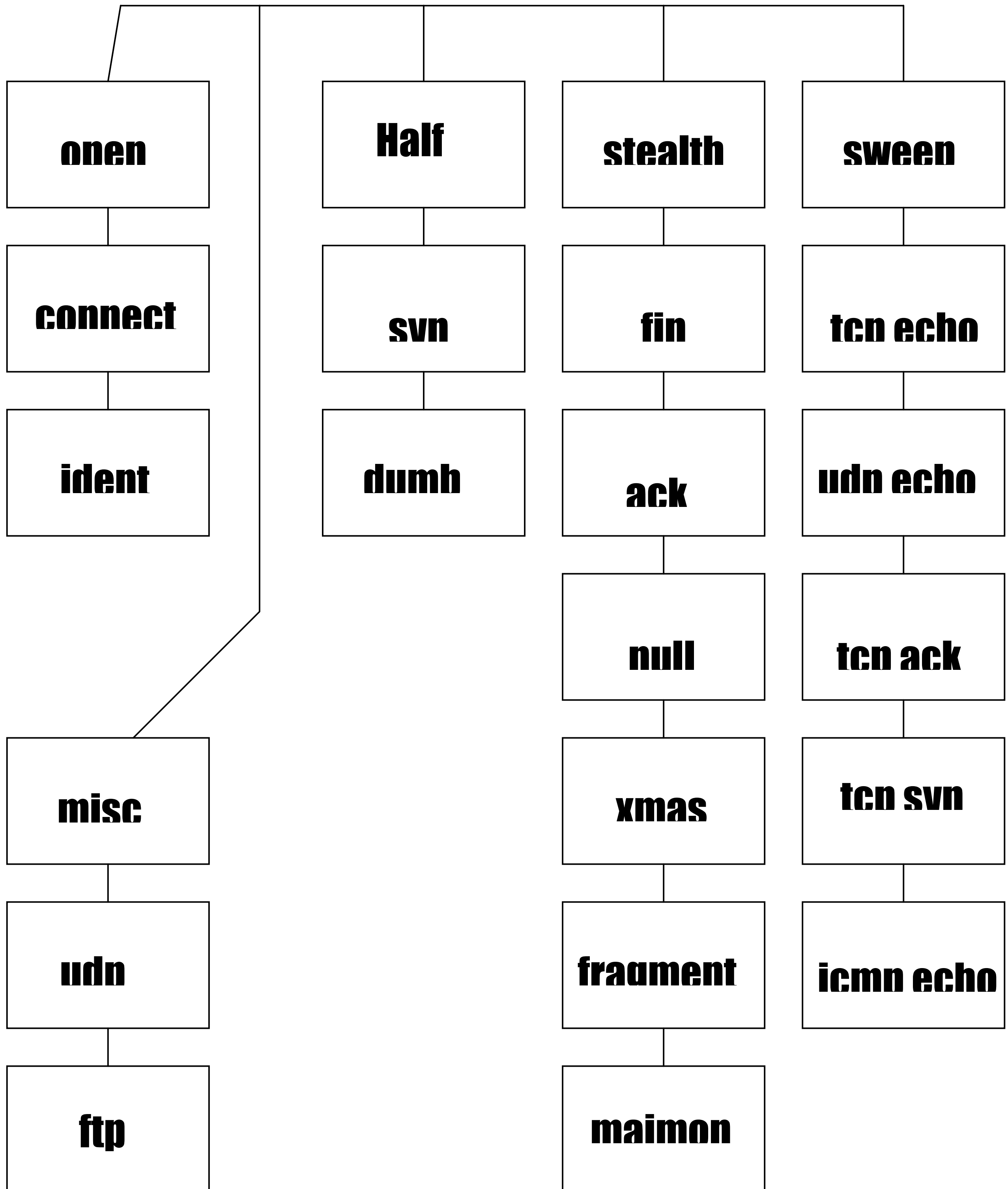
TCP ping scan: 80 포트에 SYN 패킷을 보낸다.

misc

FTP bounce scan: ftp protocol 의 헛점을 이용해서 방화벽 내부의 포트가 열린게 있는지 확인하는 방법이다.

UDP scan: 닫힌 udp 패킷을 보내면 icmp port not reachable 에러 패킷으로 반응한다.

scan 구성도



2
open scan

open scan method

open scan 기술은 쉽게 탐지할 수 있고 걸러낼 수 있다. 이 스캔 방법은 상대 시스템에 전형적인 TCP/IP Threeway handshake 를 사용해서 접속한다. 연결을 만들 때, 다음과 같은 flag를 설정하는 순서로 작동한다.

```
client -> SYN
server -> SYN|ACK
client -> ACK
```

위의 예는 초기 연결 요청에 SYN|ACK로 답하는 포트의 예를 들었다. 이 반응은 패킷을 보낸 포트가 LISTENING 즉 열려있다는 걸 의미한다. 모든 handshake가 일어나면, 클라이언트가 다른 포트를 검사하기 위해서 새로운 소켓을 만들어야 하므로 연결할 수 있는 최대 포트에 연결한 다음, 그 연결을 종료시킨다. 반대로 닫힌 포트에선, 아래와 같은 반응을 한다.

```
client -> SYN
server -> RST|ACK
client -> RST
```

RST|ACK flag는 클라이언트의 연결 요청을 취소했다는 걸 의미한다. 즉 포트는 LISTENING 상태가 아니며, 닫혀있다는 얘기이다.

connect scan

이 방법은 connect() 시스템 콜을 사용하며, 열려지거나 닫힌 포트를 거의 즉시 알 수 있다. connect() call 반환 값이 true 이면, 포트는 열려 있고 그렇지 않으면 포트는 닫혀있다.

상대 시스템에 Three-way handshake를 하기 때문에, 상대에게 우리 시스템을 속이는게 불가능하다. 덧붙이자면, 우리 시스템의 정확한 위치를 조작하는게 불가능하다는 말이다. 속이는 연결을 하려면 정확한 sequence 값과 데이터 전달을 작동시키는 정확한 flag를 필요로 하기 때문이다.

이 기술은 시스템으로 입력되는 트래픽 중에서 쉽게 탐지 될 수 있다. 이것은 모든 연결을 다 하기 때문이다, 따라서 IDS 와 방화벽 모두 이러한 스캔 공격을 탐지 할 수 있고 막을 수 있다. 그러나, connect() 방법이 three way handshake를 사용하므로 이 스캔의 결과는 포트가 열렸는지 닫혔는지 정확하게 파악할 수 있다.

장점: 빠르고 정확하며 특별한 권한이 필요없음.
단점: 쉽게 탐지되며 기록됨.

Connect scan simple code

get_connect_scan

실제 함수의 코드가 아닌 간단하게 줄인 코드를 적었다.

```
int portlookup[65536]; // 포트 번호로 실제 portlist 위치를 찾을 수 있는 배열

  PORTINFO *cur,*daum;

    // select 에 사용할 값들을 초기화 한다.
  FD_ZERO( &fds_read);
  FD_ZERO( &fds_write);
  FD_ZERO( &fds_except);

    if ( 사용자가 동시에 검사할 포트 수를 지정한다면 ) {
         ss.max_width = op.max_parallelism; // 사용자가 입력한 값을 사용한다.
  }

  ss.max_width = 실제 시스템이 동시에 열수 있는 소켓수를 구한다.

  if ( 초기에 포트를 열수 있는 값이 시스템이 동시에 여는 값보다 크다면 )
        init_packet_width = ss.max_width; // 시스템이 동시에 여는 값으로 바꾼다.

  시스템이 사용할 수 있는 이상적인 소캣 값을 지정한다.
  ss.numqueries_ideal = init_packet_width;

    // 사용할 포트를 초기화 한다.
  for ( i = 0 ; i < portcnt ; i++ ) {
        portlist[i].state = PORT_FRESH;
        portlist[i].portno = portarray[i];
        portlist[i].trynum = 0;
        portlist[i].prev = i-1;

            // 제일 마지막 포트 일 경우 next 값으로 -1 을 사용한다.
        if (i < portcnt -1 ) portlist[i].next = i+1;
            else portlist[i].next = -1;
    portlookup[portarray[i]] = i;
            // 사용할 소켓 배열은 초기화를 한다.
```

```
        portlist[i].sd[0] = portlist[i].sd[1] = portlist[i].sd[2]= -1;
}

  // 검사할 포트들을 testportlist 로 옮긴다.
testportlist = &portlist[0];

memset( &mysock , 0, sizeof( struct sockaddr_in));
mysock.sin_addr.s_addr = destaddr.s_addr;
mysock.sin_family= AF_INET;

do {

        // 연결이 많이 실패해 시도가 많아지면 delay 값을 증가시킨다.
    if ( tries > 3 && tries < 10 ) {
            senddelay += 10000 * ( tries - 3);
    } else if ( tries >= 10 ) {
            senddelay += 75000;
    }

        // senddelay 값이 200000 을 넘을 경우
        // 시스템이 사용할 수 있는 소켓의 수를 5나 더 작은 값으로 줄인다.
    if ( senddelay > 200000) {
            ss.max_width = MIN( ss.max_width, 5);
    }

        // 예측한 시각을 초과 할 경우 TIMEDOUT 부분으로 간다.
    if ( timedout )
            goto TIMEDOUT;


    while ( testportlist != NULL ) { // 검사할 포트가 남아 있으면

            gettimeofday( &now, NULL); // 시작 시각을 now에 저장한다.

                // 예측한 시각을 초과할 경우 timedout 부분을 SETTING 한다.
            if ( op.host_timeout && ( TIME_MSEC_SUB( now, host_timeout) >= 0 )) {
                    timedout = SET;
                    goto TIMEDOUT;
            }

                // testportlist 에 있는 포트들을 차례로 반복한다.
            for ( cur = testportlist ; cur ; cur = daum ) {

                        // 다음 포트 위치를 저장하고 다음 포트 위치가 끝 부분의 경우 NULL을 넣는다.
                    daum = ( cur->next > -1) ? &portlist[cur->next] : NULL;

                    check_firewallmode(); // 현재까지의 포트 상황으로 방화벽 유무를 검사한다.

                        // 현재 포트의 상태가 검사 중이면
                    if ( cur->state == PORT_TESTING) {
                                // 보낸 패킷이 예측 시각을 초과하면
                            if ( TIME_SUB( now, cur->sent[ cur->trynum]) > tmout.timeout) {
                                        // 방화벽 관련 내용이 작동 되는지 검사한다.
                                    if ( cur->trynum > 1 || ( cur->trynum > 0 && fi.active)) {

                                            fi.nonresponse++;
                                            cur->state = PORT_FIREWALLED;
                                    }

                                    // 패킷이 소멸했으므로 이 포트는 리스트에서 삭제한다.
                                    if ( cur->prev > -1) portlist[ cur->prev].next = cur->next;
                                    if ( cur->next > -1) portlist[ cur->next].prev = cur->prev;

                                    // 포트가 제일 앞의 포트 일 경우
                                    if ( cur == testportlist)
```

```
					testportlist = ( cur->next > 0) ? &portlist[cur-
>next] : NULL;

					// 방화벽 포트 리스트에 이 포트를 설정한다.
				if ( !fwportlist) fwportlist = cur;
				else {
					/* insert node at firewall port list */
					cur->next = (fwportlist - portlist) /
sizeof(PORTINFO);

					fwportlist = cur;
					portlist[ cur->next].prev = (cur - portlist) /
sizeof(PORTINFO);
				}


					// 방화벽 포트 검사할때 사용하도록 다시 설정한다.
				for ( i = 0 ; i<= cur->trynum ; i++) {
					if ( cur->sd[i] >= 0) {
						socklookup[ cur->sd[i]] = NULL;
						FD_CLR( cur->sd[i], &fds_read);
						FD_CLR( cur->sd[i], &fds_write);
						FD_CLR( cur->sd[i], &fds_except);
						close( cur->sd[i]);
						cur->sd[i] = -1;
						ss.numqueries_outstanding--;
					}
				}
			} else { // 보낸 패킷이 예측 시간 안에 도달하면

				int victim;

					// 사용자가 delay 값을 설정했다면
				if ( op.scan_delay) enforce_scan_delay( NULL);

					// 시도 횟수를 증가 시킨다.
				cur->trynum++;

					// 현재 보낸 패킷의 시간대를 기록한다.
				gettimeofday( &cur->sent[cur->trynum], NULL);
				now = cur->sent[ cur->trynum];

					// 외부에 돌아오지 않은 패킷수가 동시에 보낼 패킷수보다 많으
면

				if ( ss.numqueries_outstanding >= ss.max_width) {
					victim = -1;

					for ( i = 0; i< cur->trynum; i++)
						if ( cur->sd[i] >= 0 ) {
							victim = i;
							break;
						}
					if ( victim == -1 )
						fatal("Illegal situation in connect scan");

							// 전에 실패한 소켓들은 닫는다.
					FD_CLR( cur->sd[victim], &fds_read);
					FD_CLR( cur->sd[victim], &fds_write);
					FD_CLR( cur->sd[victim], &fds_except);
					close( cur->sd[victim]);
				} else {
					// 외부에 나가있는 패킷의 수를 증가시킨다.
					ss.numqueries_outstanding++;
				}

					// 소켓을 설정한다.
```

```
                    res = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP);
                    if ( res == -1) pfatal("Socket troubles in connect scan");
                              // 동시에 여러 포트를 열기위해서 block 모드를 해제한다.
                    unblock_socket( res);
                    init_socket( res);
                    mysock.sin_port = htons( cur->portno);
                    cur->sd[ cur->trynum] = res;

                              // connect 연결을 한다.
                    res    =    connect(    res,    (struct    sockaddr    *)&mysock,
sizeof( struct sockaddr));

                              socklookup[ res] = cur; // 소켓의 주소로 포트를 찾아내
도록 설정

                    if ( res != -1) { // 연결이 되었을 경우
                         update_port( cur , PORT_OPEN);
                    } else {
                         switch(errno) {
                              case EINPROGRESS: // 연결이 진행중인 경우
                              case EAGAIN:
                                   if ( maxsd < cur->sd[ cur->trynum])
                                        maxsd = cur->sd[ cur->trynum];

                                   FD_SET(       cur->sd[       cur->trynum],
&fds_read);
                                   FD_SET(       cur->sd[       cur->trynum],
&fds_write);
                                   FD_SET(       cur->sd[       cur->trynum],
&fds_except);
                                   break;
                              default: // 그 외의 경우 - 최근 방화벽
                                   if (!connecterror) {
                                        connecterror++;
                                        fprintf(  stderr,  "Strange  error
from connect (%d):", errno);
                                        fflush( stdout);
                                        perror("");
                                   }
                              case ECONNREFUSED: // 연결이 거부 되었을 경우
                                   update_port( cur, PORT_CLOSED);
                                   break;

                         }
                    }

                    if (senddelay) usleep( senddelay);
               }
          } else { // 포트 상태가 PORT_TESTING 이 아닐 경우
               if ( cur->state != PORT_FRESH) {
                    fatal("State mismatch!! %d", cur->state);
               }

                    // 충분하게 많은 패킷을 시스템에 보내고 기다린 다면 break
               if ( ss.numqueries_outstanding >= (int) ss.numqueries_ideal) break;
               if ( op.scan_delay) enforce_scan_delay( NULL);


               cur->state = PORT_TESTING;
               cur->trynum = 0;

               printf("Insert Data PORT_TESTING portno %d state %d\n", cur->portno,
cur->state); //
               ss.numqueries_outstanding++;
               gettimeofday( &cur->sent[0], NULL);
```

```
                        res = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP);
                        if (res == -1) pfatal("Socket troubles in connect_scan");

                        socklookup[ res] = cur; // 소켓의 주소로 포트를 찾아내도록 설정

                        unblock_socket( res);
                        init_socket( res);
                        mysock.sin_port = htons( cur->portno);
                        cur->sd[ cur->trynum] = res;
                        res  =  connect(  res,  (struct  sockaddr  *)&mysock,  sizeof(  struct
sockaddr));

                        if ( res != -1) { // 연결이 성공할 경우 포트가 열려있는 경우
                                update_port( cur, PORT_OPEN);
                        } else {
                                switch(errno) {
                                        case EINPROGRESS: // 연결이 진행중일 경우
                                        case EAGAIN:
                                                if ( maxsd < cur->sd[ cur->trynum])
                                                        maxsd = cur->sd[ cur->trynum];
                                                FD_SET( cur->sd[ cur->trynum], &fds_read);
                                                FD_SET( cur->sd[ cur->trynum], &fds_write);
                                                FD_SET( cur->sd[ cur->trynum], &fds_except);
                                                break;
                                        default: // 최근 방화벽에서 나타남
                                                if (!connecterror) {
                                                        connecterror++;
                                                        fprintf(  stderr,  "Strange  error  from
connect (%d):", errno);

                                                        fflush( stdout);
                                                        perror("");
                                                }
                                        case ECONNREFUSED: // 연결이 거부 될 경우
                                                update_port( cur, PORT_CLOSED);
                                                break;
                                }
                        }

                        if (senddelay) usleep( senddelay);
                }
        } /* for */

        ss.alreadydecreasedqueries = 0;

        get_connect_results(); // 보낸 패킷들에서 온 패킷을 보고 열리고 닫힌 걸 결정하는 함수

        if ( timedout)
                goto TIMEDOUT;
}

if ( ss.numqueries_outstanding != 0)
        fatal(" Port count error in connect_scan");

   // 방화벽 으로 막아진 걸로 설정된 포트가 있다면
if ( fwportlist) {
            // 다시 검사
        if ( tries == 0 || ss.changed) {
                testportlist = fwportlist;
                for ( cur = testportlist ; cur ; cur= daum) {
                        daum = ( cur->next > -1) ? &portlist[ cur->next] : NULL;
                        cur->state = PORT_FRESH;
                        cur->trynum = 0;
                        cur->sd[0] = cur->sd[1] = cur->sd[2] = -1;
                }
```

```
                        fwportlist = NULL;
                } else { // 방화벽이 존재하는 걸로 알려질 경우
                                // 방화벽 리스트의 모든 포트를 방화벽으로 막힌 걸로 설정
                        for ( cur = fwportlist ; cur ; cur = daum) {
                                daum = ( cur->next > -1) ? &portlist[ cur->next] : NULL;
                                myportlist.add( cur->portno, IPPROTO_TCP, PORT_FIREWALLED);
                        }
                        testportlist = NULL;
                }
                tries++;
        }

        ss.numqueries_ideal = init_packet_width;
    } while( testportlist && tries < 20); // 검사할 모든 포트를 했거나 시도가 20번 이상을 넘을 경우

    if ( tries == 20 ) // 20번이상 패킷을 보냈는데 실패할 경우는 네트워크 이상
        error("WARNING: GAVE UP SCAN AFTER 20 RETRIES");

TIMEDOUT:
    return;
}
```

get_connec_result 부분

```
  PORTINFO *cur;

  do {
    timeout.tv_sec = 0;
    timeout.tv_usec = 20000;
    selectedfound = 0;

     // Timeout 시각이 지나면 Timeout bit set
    if (op.host_timeout) {
      gettimeofday(&tv, NULL);
      if (TIME_MSEC_SUB(tv, host_timeout) >= 0) {
        timedout = SET;
        return;
      }
    }

     // select
    selectres = select( maxsd+1, &fds_read, &fds_write, &fds_except, &timeout);

     // select 함수가 가지고 있는 값이 크거나 소켓의 최고값을 넘지않을 동안 모든 소켓을 차례로 검사
    for(sd=0; selectedfound < selectres && sd <= maxsd; sd++) {

      cur = socklookup[sd];

      if (!cur) continue;

      trynum = -1;
       // 소켓에 반응이 있을 경우
      if  (FD_ISSET(sd, &fds_read)  || FD_ISSET(sd, &fds_write) ||
            FD_ISSET(sd, &fds_except)) {
        for(i=0; i < 3; i++)
          if (cur->sd[i] == sd) {
            trynum = i;
            break;
          }

        if (FD_ISSET(sd, &fds_read)) {
          selectedfound++;
        }
        if (FD_ISSET(sd, &fds_write)) {
          selectedfound++;
```

```
            }
            if (FD_ISSET(sd, &fds_except)) {
               selectedfound++;
            }

            assert(trynum != -1);

            if (getsockopt(sd, SOL_SOCKET, SO_ERROR, (char *) &optval, &optlen) != 0)
               optval = errno; /* Stupid Solaris ... */

            switch(optval) {
            case 0:
                  // Linux 의 경우는 직접 0 byte 패킷을 보내는 검사를 해야 한다.
               if (!FD_ISSET(sd, &fds_read)) {
                  res = send(cur->sd[trynum], "", 0, 0);

                  if (res < 0 ) {
                     update_port( cur, PORT_CLOSED);


                  } else {
                     if (getpeername(sd, (struct sockaddr *) &sin, &sinlen) < 0) {
                        pfatal("error in getpeername of connect_results for port %hu", cur-
>portno)
;

                     } else {
                        if (cur->portno != ntohs(sin.sin_port)) {
                           error("Mismatch!!!! we think we have port %hu but we really have %hu",
cur-
>portno, ntohs(sin.sin_port));
                        }
                     }

                     if (getsockname(sd, (struct sockaddr *) &sout, &soutlen) < 0) {
                        pfatal("error in getsockname for port %hu", cur->portno);
                     }
                     if (htons(sout.sin_port) == cur->portno) { // 잘못된 패킷을 받았는지 검사
                        update_port( cur, PORT_CLOSED);
                     } else {
                        update_port( cur, PORT_OPEN);

                     }
                  }
               } else {
                  update_port( cur, PORT_OPEN);
               }
               break;
            case ECONNREFUSED: // 연결이 거부 당했을 경우
               update_port( cur, PORT_CLOSED);
               break;
            case EHOSTUNREACH:
            case ETIMEDOUT:
            case EHOSTDOWN: // 실제 시스템이 다운 되었거나 방화벽이 있을 경우
               update_port( cur, PORT_FIREWALLED);
               break;
            case ENETDOWN:
            case ENETUNREACH:
            case ENETRESET:
            case ECONNABORTED:
               snprintf(buf, sizeof(buf), "Strange SO_ERROR from connection (%d) -- bailing scan",
optval)
;
               perror(buf);
               return;
               break;
```

```
            default:
                snprintf(buf, sizeof(buf), "Strange read error (%d)", optval);
                perror(buf);
                break;
            }
        } else continue;
    } /* for */
// 밖으로 내보낸 패킷이 다 돌아오고 select 가 가진 소켓이 결과를 다 처리하거나 timeout 이 되었을 경우
} while(ss.numqueries_outstanding > 0 && selectres > 0);
```

Reverse ident scan

이 기술은 ident/auth 데몬의 반응값을 사용한다. 113번 포트는 작동되는 프로세스의 권한을 묻는데 사용된다. 이런 방법으로 root 권한으로 도는 데몬을 알아낼 수 있다. 침입자는 root권한의 포트의 취약점을 찾거나 다른 의심스러운 행동을 할 수 있다. nobody 권한으로 돌아간다면, 제한된 권한을 가지고 있기 때문에 공격자가 관심을 덜 느낀다. 많은 수의 사용자가 identd 데몬이 아래에 나온 사소하지만 사적인 정보들을 유출 한다는 걸 알지 못한다.

```
 * user info
 * entities
 * objects
 * processes
```

비록 idend가 사용하는 프로토콜이 인증 메카니즘처럼 보이지만, 이 프로토콜은 이렇게 사용할 목적으로 만들지는 않았다. RFC 를 보면 "TCP 연결에 몇가지 검사 정보를 제공할 뿐이다."라 쓰여있다. 말할 필요도 없이 접근 제어 서비스로 사용되는 것도 아니며, 호스트 / 사용자명 인증을 너무 믿어도 안 된다.

RFC 1413 을 보면 EBNF 형식의 문법을 찾을 수 있다.

```
FORMAL SYNTAX

   <request> ::= <port-pair> <EOL>
   <port-pair> ::= <integer> "," <integer>
   <EOL> ::= "015 012"  ; CR-LF End of Line Indicator, octal \r\n equivalents
   <integer> ::= 1*5<digit> ; 1-5 digits.
```

이 문법대로 만든 데이터를 ident/auth 포트에 보내면, 그 포트와 관련된 프로세스에 대한 정보를 얻을 수 있다. 물론 먼저 연결해야 한다.

장점: 빠르며, 권한이 별로 필요 없고, 실제 서비스 정보를 얻을 수 있음

단점: 쉽게 탐지됨

Reverse ident scan simple conde

nmap.c

```
int getidentinfoz(struct in_addr target, u16 localport, u16 remoteport, char *owner, int ownersz)

          // 소켓을 열고

          // connect
          res = connect(sd, (struct sockaddr *) &sock, sizeof(struct sockaddr_in));

          // identd 에 전달할 문자열
          snprintf(request, sizeof(request), "%hu,%hu\r\n", remoteport, localport);

          // 값 전달
          write(sd, request, strlen(request)

          // 값 출력
          read(sd, response, sizeof(response)

          // ':' 단위로 끊어서 분석
          strchr(response, ':')
          p++;
          q = strtok(p, " :")
}
```

3

half open scan

half open scan method

'half open' 이란 말은 three-way handshake 가 이뤄지기 전에 클라이언트가 연결을 끊어버리기 때문에 사용하는 말이다. 그래서 이 스캔은 connection 연결만 탐지하는 IDS에는 기록이 남지 않는다. 그리고 open/closed 포트 인식을 사용하므로 정확한 결과를 얻을 수 있다.

SYN scan

이 스캔은 Three way handshake 에서 ACK 를 보내는 대신 연결을 끊어 버리는 점만 제외하면, TCP connect scan과 비슷하다. 쉽게 이 기술을 말한다면, connect scan 을 반 정도만 진행하면 된다.

```
client -> SYN
server -> SYN|ACK
client -> RST
```

시스템의 포트가 열려 있으면, SYN|ACK flag로 답한다. RST flag는 커널의 TCP/IP 스택 코드에서 자동으로 보내므로 클라이언트에서는 RST 패킷을 보낼 필요가 없다. 이와 다르게 , 닫힌 포트에서는 RST|ACK 로 응답한다.

```
client -> SYN
server -> RST|ACK
```

보다시피 이 RST|ACK 의 조합은 열린 포트가 아니라는 점을 알려준다.
이 기술이 많은 IDS에서 탐지하기 쉬운 건, 많은 SYN 패킷을 보내는 서비스 공격 중의 하나인 SYN flood와 관련되어 있기 때문이다. 괜찮은 IDS (TCP wrappers, SNORT, Courtney, iplog)에서는 이런 half-open 스캔을 기록할 수 있다. 그래서 최근에는 IP를 숨길 수 있었던 장점도 빛이 많이 바랬다.

장점: 빠르고, 믿을만 하며, Three-way handshake를 피할 수 있음

단점: root 권한이 필요하며 많은 SYN 시도는 방화벽에서 막음

SYN scan simple code

scan_engine.c

```
void pos_scan(struct hoststruct *target, u16 *portarray, int numports, stype scantype)

u32 sequences[3];

   // raw socket 사용
        if ((rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0 )

   broadcast_socket(rawsd);

   // sequece 값으로 random 값 사용
   get_random_bytes(sequences, sizeof(sequences));

   // pcap 설정
   pd = my_pcap_open_live(target->device, 100,  (o.spoofsource)? 1 : 0, 20);

   flt_srchost = target->host.s_addr;
   flt_dsthost = target->source_ip.s_addr;

   p = strdup(inet_ntoa(target->host));

   // 상대 시스템에서 오는 모든 패킷들을 검사하는 필터 설정
   snprintf(filter, sizeof(filter), "(icmp and dst host %s) or (tcp and src host %s and dst
host %s)", inet_ntoa(target->source_ip), p, inet_ntoa(target->source_ip));
   free(p);

   set_pcap_filter(target, pd, flt_icmptcp, filter);

   scanflags = TH_SYN;

   starttime = time(NULL);

   // syn scan 이므로 ack 값을 0
   ack_number = 0;


   send_tcp_raw_decoys(rawsd, &target->host, o.magic_port +
                                          tries * 3 + current->trynum,
                                                         current->portno,
                                                         sequences[current->trynum],
                                                         ack_number, scanflags, 0, NULL, 0,
                                                         o.extra_payload,
                                                         o.extra_payload_length);

   // 보낸 syn packet 에 대한 반응을 수집
   get_syn_results(target, scan, &ss, &pil, portlookup, pd, sequences, scantype);

}

get_syn_result

get_syn_results(struct hoststruct *target, struct portinfo *scan,
          struct scanstats *ss, struct portinfolist *pil,
```

```
            int *portlookup, pcap_t *pd, u32 *sequences,
            stype scantype)

ip = (struct ip*) readip_pcap(pd, &bytes, target->to.timeout))) {

// 맞는 패킷인지 확인
if (ip->ip_src.s_addr == target->host.s_addr && ip->ip_p == IPPROTO_TCP) {
        tcp = (struct tcphdr *) (((char *) ip) + 4 * ip->ip_hl)
        i = ntohs(tcp->th_dport);

    newport = ntohs(tcp->th_sport);

    // syn 이나 ack 만 설정 될 경우
    if (ip->ip_src.s_addr == target->source_ip.s_addr && ((tcp->th_flags == TH_ACK) || (tcp->th_flags
== TH_SYN))) {
              continue;
    }

    current = &scan[portlookup[newport]];

    if (tcp->th_flags & TH_RST) {
                newstate = PORT_CLOSED;
    } else if ((tcp->th_flags & (TH_SYN|TH_ACK)) == (TH_SYN|TH_ACK)) {
                newstate = PORT_OPEN;
    }

        return;
}
```

IDLE scan

IDLE(dumb) scan은 대부분의 운영체제 TCP/IP 의 구현 특징을 사용한 좀 애매한 스캔 방법이다. 이 방법은 antirez 가 발견했고, 그가 기술 적인 내용을 bugtraq에 올렸다. dummy source 로 다른 시스템을 사용하지만, 스캔 구현의 기본 원리는 SYN scan 방법을 사용한다..

깊게 들어가기 전에 dumb 호스트라는 말의 의미를 아는게 중요하다. bastion host 호스트와 다르게 이 호스트는 어떤 트래픽도 외부로 보내지 않는다. 그런 모습을 보고 저런 특징적인 이름을 지었다. 이 호스트 중에 하나는 많은 일도 해야 하며 호스트 sweeping 도 해야 한다. 스캔 방법은 쓸모 보다는 문제가 더 많지만, 방법이 독창적이며 뛰어나다. 다른 호스트가 간접적으로 스캔을 하게 해서 실제 IP를 숨긴다.

세 개의 호스트가 있는 시나리오를 생각해 보자.

* A -> 공격자 호스트
* B -> dumb 호스트
* C -> 목표 호스트

호스트 A에서는 ID 필드 분석을 위해서 여럿의 PING 을 보낸다. 호스트 B의 IP 헤더 부분을 캡슐화 시켜서. PING 신호를 보낼 때마다 dumb 호스트에서 오는 패킷의 ID 값이 일정하게 증가한다.

```
60 bytes from BBB.BBB.BBB.BBB: seq=1 ttl=64 id=+1 win=0 time=96 ms
60 bytes from BBB.BBB.BBB.BBB: seq=2 ttl=64 id=+1 win=0 time=88 ms
60 bytes from BBB.BBB.BBB.BBB: seq=3 ttl=64 id=+1 win=0 time=92 ms
```

이때 호스트 A에서 B의 주소를 사용해서 호스트 C 로 SYN 패킷을 보낸다. 원격 포트는 공격자가 열려있는지 닫혀있는지 확인하고픈 포트로 설정한다. 호스트 C에서는 B로 다음 2가지 방법으로 반응을 한다.

-> SYN|ACK 로 반응하는 건 LISTENING , 열린 포트를 의미한다. 이 패킷을 받은 호스트 B에서는 RST flag가 설정된 패킷을 보낸다.
-> RST|ACK 로 반응하는 건 닫힌 포트를 의미한다. 일반적인 SYN 스캔 방법처럼 호스트 B는 이 패킷을 무시하고 아무 일을 하지 않는다.

그러면 호스트 A에서는 호스트 B의 반응을 어떻게 알 수 있을까? 목적하는 서버의 포트가 열렸는지 안 열렸는지 확인하기 위해서, 호스트 A에선 여러개의 PING을 보내고 답으로 오는 ID 필드의 변화를 분석한다. 물론 이때도 C 호스트로 IP 주소를 속인 패킷을 보낸다. 이 때 A에서 B로 보낸 PING 값의 반응으로 온 패킷을 살펴 보면 ID 값의 불규칙한 증가를 볼 수 있다.

```
60 bytes from BBB.BBB.BBB.BBB: seq=25 ttl=64 id=+1 win=0 time=92 ms
60 bytes from BBB.BBB.BBB.BBB: seq=26 ttl=64 id=+3 win=0 time=80 ms
60 bytes from BBB.BBB.BBB.BBB: seq=27 ttl=64 id=+2 win=0 time=83 ms
```

2번째와 3번째 패킷 ID 가 1 보다 큰 값이 나왔다. open port임을 가리킨다. 값의 증가는 Host B가 그동안 열린 포트에 대한 답을 했다는 말이 된다. 일반적으로, 1 증가는 , 호스트 A에서 주소지를 속인 SYN을 열린 포트에 보내기 때문에 호스트 B에서는 C에서 온 SYN|ACK 패킷에 반응해야 한다. 이 반응은 HOST A에 대한 증가한 ID field 를 가진 PING 반응을 보고 알 수 있다. 이와 반대로, C의 닫힌 포트에서 온 RST 패킷엔 호스트 B가 어떤 패킷도 보내지 않아서 응답하는 ID 필드의 변화값이 나타나지 않는다.

```
60 bytes from BBB.BBB.BBB.BBB: seq=30 ttl=64 id=+1 win=0 time=90 ms
60 bytes from BBB.BBB.BBB.BBB: seq=31 ttl=64 id=+1 win=0 time=88 ms
60 bytes from BBB.BBB.BBB.BBB: seq=32 ttl=64 id=+1 win=0 time=87 ms
```

이처럼 ID 필드는 1이 증가해야 한다. 이것이 dumb 호스트라 불릴 필요가 있는 이유이다. B 호스트가

트래픽이 높은 경우에 다른 값을 낼 수 있으므로, 트래픽이 없을 때 조사해야 한다. SYN 패킷을 보내는 방법 말고도, 다양한 scan 방법을 dumb 호스트를 사용해서 해 볼 수 있다. 스캔 방법은 꼭 SYN 에만 정해진것이 아니라 호스트 B가 호스트 A의 포트에 답하는 어떤 방법도 사용할 수 있다.

IDLE scan simple code

(나중에 추가 예정)

4

stealth scan

stealth scan method

스텔스 스캔의 정의는 Chris Kluaus의 " Stealth Scanning: Bypassing Firewalls/SATAN Detectors" 에서 나왔다. 이 단어는 "half open" 스캐닝처럼 IDS 를 피하거나 기록이 되지 않는 기술을 말한다. 그러나 요즘에는 아래 몇 가지와 연관된 스캔 방법을 의미한다.

* flag를 하나씩 켜서 작동시키는 경우
* 아무 flag를 켜지 않을 경우
* 모든 flag를 켤 경우
* 라우터나 방화벽 필터를 우회할 경우
* 네트워크 환경에서 가끔 나타나는 경우
* 다양한 패킷이 여러 비율로 나타나는 경우

이 장에서 설명하는 모든 스캔 방법이 아무 반응을 하지 않는 포트를 열린 포트로 여기는 방법을 사용한다.

MAIMON(SYN|ACK) scan

이 기술은 오늘날까지 무시된 기술이다. 특이하게, 이 스캔의 이론은 SYN 방법과 다르지 않다. half-open 스캔의 첫번째 단계의 내용을 잘라서 보면. 다음과 같은 반응을 한다.

```
client -> SYN|ACK
server -> RST
```

위에 나타난 flag는 포트가 닫혀 있음을 뜻한다. 어떠한 최초의 SYN 패킷도 보내지 않았기 때문에, 즉시 반응을 한다. 다시 말하자면 SYN 없이 SYN|ACK를 보냈기 때문에 에러가 있는 걸로 확인하고, RST 패킷을 보낸다. 이와 반대로 열린 포트에서는 RST 패킷으로 반응하는 걸 찾을 수 없다.

```
client -> SYN|ACK
server -> -
```

보는 바와 같이, 서버는 열린 포트에 보내는 SYN|ACK 패킷을 무시한다. 이런 경우, 우리에게 오지 않는 패킷은 false positve를 일으킨다. SYN|ACK 패킷을 보내고 패킷 필터나 , 방화벽 또는 타임아웃 때문에 반응을 못 받는 패킷이 있다면, 스캐너는 그 포트에 대한 false positive를 일으킨다. 따라서 이러한 스캔 방법은 이런 이유로 connect scan 스캔보다 믿음이 덜 간다. 답하지 않는 포트를 열린 포트로 여기는 가정을 inverse mapping 이라고 한다.

장점: 빠르며 간단한 IDS와 방화벽, 그리고 Three way handshake 를 피할 수 있음
단점: 신뢰성이 떨어짐. (false positive 가 많다.)

Maimon scan simple code

scan_engine.c

```
void super_scan(struct hoststruct *target, u16 *portarray, int numports, stype scantype)

rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)

broadcast_socket(rawsd);

pd = my_pcap_open_live(target->device, 92,  (o.spoofsource)? 1 : 0, 10);

flt_srchost = target->host.s_addr;
flt_dsthost = target->source_ip.s_addr;
flt_baseport = o.magic_port;

p = strdup(inet_ntoa(target->host));
// 필터 설정
snprintf(filter, sizeof(filter), "(icmp and dst host %s) or (tcp and src host %s and dst host %s and ( dst port %d or dst port %d))", inet_ntoa(target->source_ip), p,    inet_ntoa(target->source_ip), o.magic_port , o.magic_port + 1);

// 특정 시각 안에 반응이 오지 않으면 열린 포트로 설정
if ( TIMEVAL_SUBTRACT(now, current->sent[current->trynum]) > target->to.timeout)
        current->state = PORT_OPEN;

if (scantype == MAIMON_SCAN) scanflags = TH_FIN|TH_ACK;

send_tcp_raw_decoys(rawsd, &target->host, i,
                 current->portno, 0, 0, scanflags, 0, NULL, 0,
                 o.extra_payload, o.extra_payload_length);

while (!timedout && numqueries_outstanding > 0 && ( ip = (struct ip*) readip_pcap(pd, &bytes, target->to.timeout)))

if (tcp->th_flags & TH_RST) newstate = PORT_CLOSED;

// Super_scan 을 사용할 경우 반응이 오지 않으면 open 으로 여기는데
// 이때 방화벽 설정이 엄격한 시스템의 경우 다 열려있는 것으로 나온다.
// 그래서 25개 이상의 Port 가 열린 경우는 엄격한 방화벽이 있는 것으로 확인하고
```

```
// 모든 포트를 PORT_FIREWALLED 로 한다.

if (numports > 25) {

    for(portno = 0; portno < 65536; portno++)
      {
        current_port_tmp = lookupport(&target->ports, portno, IPPROTO_TCP);
        if (current_port_tmp) {
          assert(current_port_tmp->state == PORT_OPEN);
          current_port_tmp->state = PORT_FIREWALLED;
          target->ports.state_counts[PORT_OPEN]--;
          target->ports.state_counts[PORT_FIREWALLED]++;
          target->ports.state_counts_tcp[PORT_OPEN]--;
          target->ports.state_counts_tcp[PORT_FIREWALLED]++;
        }
      }
      }
    }
}
```

FIN scan

FIN 스캔 방법은 포트를 찾는데 inverse mapping 을 사용한다. 이 기법은 BSD 기반의 운영체제에서 사용하는 네트워크 코드에 기반을 둔 기술이라서 모든 운영체제에 사용할 수 없다. 이상적으로 FIN 패킷을 보내면 닫힌 포트는 RST 패킷을 보낸다. 이와 비교해 , 열린 포트에서는 패킷을 보내지 않는다. FIN 패킷을 보내서 정확하게 대답을 받을 수 없으면 inverse mapping으로 열린 포트임을 알 수 있다. 열렸는지 닫혔는지 확인하는 순서는 아래에 설명한다.

```
client -> FIN
server -> -
```

서버에서 어떤 대답을 받을 수 없다면, 그 포트는 열린 포트이다. 서버 운영체제는 그 포트에서 작동하는 서비스에 도달한 FIN 패킷을 버린다. 이에 비해 닫힌 포트엔 RST 패킷으로 응답을 한다. 즉 그 포트에서 어떤 서비스도 하지 않는다면 , FIN 패킷을 보내면 서버에선 RST 로 응답한다는 말이다.

```
client -> FIN
server -> RST
```

열린 포트를 검사 할 수 있는 2가지 방법이 있다. 첫번째는 닫힌 포트의 응답을 받아서 원래 보낸 포트에서 빼고 남는 포트가 열린 포트이다. 예로, 3개의 패킷을 포트 1, 2, 3으로 보냈다면 1과 3에서 RST 패킷을 받았다면 원래 패킷을 보낸 리스트인 1,2,3 과 비교해서 1,3 의 신호를 받아서 2를 추출할 수 있다. 즉 2가 열린 포트이다.

두번째 검사 방법은 패킷의 반응 시간을 조사하는 것이다. 어느 패킷이 Timeout 될 때까지 패킷이 도달하지 않았다면 , 그 포트는 아마 열려있다고 생각할 수 있다. 여기서는 false positive에 대한 검사를

더 해야 한다. 느린 네트웍이나 많은 트래픽 때문에 방화벽이나 , 패킷 필터, 라우터 때문에 패킷의 반응이 애매할 수 있기 때문에 , 이런 주먹 구구식 FIN 스캔 방법은 확실하고 믿을 수 있는 검사가 될 수 없다.

장점:많은 IDS 와 Three way handshake 를 피할 수 있음.
단점: false positive가 많고 검사를 위해서 느려짐.

FIN scan simple code

scan_engine.c

```
void super_scan(struct hoststruct *target, u16 *portarray, int numports, stype scantype)

rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)
broadcast_socket(rawsd);

pd = my_pcap_open_live(target->device, 92,  (o.spoofsource)? 1 : 0, 10);


flt_srchost = target->host.s_addr;
flt_dsthost = target->source_ip.s_addr;
flt_baseport = o.magic_port;

p = strdup(inet_ntoa(target->host));
// 필터 설정
snprintf(filter, sizeof(filter), "(icmp and dst host %s) or (tcp and src host %s and dst host %s and
( dst port %d or dst port %d))", inet_ntoa(target->source_ip), p, inet_ntoa(target->source_ip),
o.magic_port , o.magic_port + 1);

// 특정 시각 안에 반응이 오지 않으면 나머지 포트를 열린 포트로 설정
if ( TIMEVAL_SUBTRACT(now, current->sent[current->trynum]) > target->to.timeout)
        current->state = PORT_OPEN;

if (scantype == MAIMON_SCAN) scanflags = TH_FIN;

if (o.fragscan) send_small_fragz_decoys(rawsd, &target->host, 0,i, current->portno, scanflags);

send_tcp_raw_decoys(rawsd, &target->host, i,
                  current->portno, 0, 0, scanflags, 0, NULL, 0,
                  o.extra_payload, o.extra_payload_length);

while (!timedout && numqueries_outstanding > 0 && ( ip = (struct ip*) readip_pcap(pd, &bytes,
target->to.timeout)))

if (tcp->th_flags & TH_RST) newstate = PORT_CLOSED;

// Super_scan 을 사용할 경우 반응이 오지 않으면 open 으로 여기는데
// 이때 방화벽 설정이 엄격한 시스템의 경우 다 열려있는 것으로 나온다.
// 그래서 25개 이상의 Port 가 열린 경우는 엄격한 방화벽이 있는 것으로 확인하고
// 모든 포트를 PORT_FIREWALLED 로 한다.

if (numports > 25) {
// 포트 상태를 PORT_FIREWALLED 로 변환
}
```

ACK scan

Uriel Maimon은 Phrack 49-15에 이 기술을 소개했다. 이 기법은 몇 몇 운영체제의 IP 계층에 존재하는 버그를 이용한다. 먼저 열린 포트에 대한 검사를 위해 ACK 패킷을 검사할 호스트에 보낸다. 여기서 반응하는 패킷을 2가지로 구분할 수 있다. TTL 필드의 평가를 보고 판단하고, 다른 하나로 WINDOW 필드를 보고 판단한다. 이 정보는 되돌아 오는 RST 패킷에서 얻을 수 있다.

TTL 값을 보는 방법은 ACK 패킷을 보내면 RST 패킷을 받지만, RST 패킷부분이 아닌 함께 오는 IP 헤더를 검사하는 것이다. 몇 몇 시스템의 닫힌 포트에서 오는 패킷은 TTL 값이 낮다. 즉 열린 포트에서 보내는 TTL 값은 64 이하이며, 다른 포트들은 더 높은 TTL 값을 가지고 있다.

```
client -> ACK
server -> RST -> (TTL <= 64)
```

실제 반응을 살펴 보면 다음과 같다.

```
packet 1: host XXX.XXX.XXX.XXX port 20: F:RST -> ttl: 70 win: 0 => closed
packet 2: host XXX.XXX.XXX.XXX port 21: F:RST -> ttl: 70 win: 0 => closed
packet 3: host XXX.XXX.XXX.XXX port 22: F:RST -> ttl: 40 win: 0 => open
packet 4: host XXX.XXX.XXX.XXX port 23: F:RST -> ttl: 70 win: 0 => closed
```

패킷2와 패킷4의 TTL 값을 살펴보면 64 보다 높다. 패킷 3의 TTL 값은 64 보다 낮은 걸 볼 수 있다. 이건 열린 포트를 의미한다.

이 스캔 방법은 방화벽의 룰을 검사할 때도 사용한다. 단순히 SYN 패킷만 막는 방화벽이라면 통과할 수 있다.

장점: 기록이 어렵다. IDS를 피할 수 있음
단점: BSD 기반의 운영체제만 되므로 운영체제에 의존.

ACK scan simple code

scan_engine.c

```
pos_scan(struct hoststruct *target, u16 *portarray, int numports, stype scantype)

rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)

broadcast_socket(rawsd);
```

```
// ISN 값 설정
get_random_bytes(sequences, sizeof(sequences));

pd = my_pcap_open_live(target->device, 100,  (o.spoofsource)? 1 : 0, 20);

flt_srchost = target->host.s_addr;
flt_dsthost = target->source_ip.s_addr;

snprintf(filter, sizeof(filter), "(icmp and dst host %s) or (tcp and src host %s and dst host %s)",
inet_ntoa(target->source_ip), p, inet_ntoa(target->source_ip));
set_pcap_filter(target, pd, flt_icmptcp, filter);

scanflags = TH_ACK;
ack_number = get_random_uint();

send_tcp_raw_decoys(rawsd, &target->host, o.magic_port +
                 tries * 3 + current->trynum,
                 current->portno,
                 sequences[current->trynum],
                 ack_number, scanflags, 0, NULL, 0,
                 o.extra_payload,
                 o.extra_payload_length);


get_syn_results(struct hoststruct *target, struct portinfo *scan,
           struct scanstats *ss, struct portinfolist *pil,
           int *portlookup, pcap_t *pd, u32 *sequences,
           stype scantype)

readip_pcap(pd, &bytes, target->to.timeout)

// Unfiltered 값을 가진다.
if (tcp->th_flags & TH_RST) newstate = PORT_UNFIREWALLED;
```

Window scan

윈도우 필드 방법을 사용하면, 윈도우 값이 0이 아닌 패킷을 가진 반응을 보인 포트가 열린 포트이다. 이 스캔은 초창기 FreeBSD , OpenBSD 와 AIX , Digital Unix 에서 사실이었지만 최근 버전에는 패치가 되었다.

```
client -> ACK
server -> RST -> WINDOW (non-zero)
```

실제 반응을 살펴 보면 다음과 같다.

```
packet 6: host XXX.XXX.XXX.XXX port 20: F:RST -> ttl: 64 win: 0 => closed
packet 7: host XXX.XXX.XXX.XXX port 21: F:RST -> ttl: 64 win: 0 => closed
packet 8: host XXX.XXX.XXX.XXX port 22: F:RST -> ttl: 64 win: 512 => open
packet 9: host XXX.XXX.XXX.XXX port 23: F:RST -> ttl: 64 win: 0 => closed
```

TTL 이 64 이더라도 주위 패킷이 64 이므로 , ACK scan은 여기에 먹히지 않는다. Window 값이 0이

아닌 포트 22가 열린 포트이다.

장점: 기록이 어렵다. IDS를 피할 수 있음
단점: BSD 기반의 운영체제만 되므로 운영체제에 의존.

Window scan simple code

scan_engine.c

```
pos_scan(struct hoststruct *target, u16 *portarray, int numports, stype scantype)

rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)

broadcast_socket(rawsd);

// ISN 설정
get_random_bytes(sequences, sizeof(sequences));

pd = my_pcap_open_live(target->device, 100,  (o.spoofsource)? 1 : 0, 20);

flt_srchost = target->host.s_addr;
flt_dsthost = target->source_ip.s_addr;

snprintf(filter, sizeof(filter), "(icmp and dst host %s) or (tcp and src host %s and dst host %s)",
inet_ntoa(target->source_ip), p, inet_ntoa(target->source_ip));
set_pcap_filter(target, pd, flt_icmptcp, filter);

// ACK flag 설정
scanflags = TH_ACK;
ack_number = get_random_uint();

send_tcp_raw_decoys(rawsd, &target->host, o.magic_port +
                 tries * 3 + current->trynum,
                 current->portno,
                 sequences[current->trynum],
                 ack_number, scanflags, 0, NULL, 0,
                 o.extra_payload,
                 o.extra_payload_length);


get_syn_results(struct hoststruct *target, struct portinfo *scan,
           struct scanstats *ss, struct portinfolist *pil,
           int *portlookup, pcap_t *pd, u32 *sequences,
           stype scantype)

readip_pcap(pd, &bytes, target->to.timeout)

// window 값 검사
if (tcp->th_flags & TH_RST) {
        if (tcp->th_win) {
       newstate = PORT_OPEN;
   } else {
       newstate = PORT_CLOSED;
   }
}
```

NULL scan

Null scan이란 이름은 TCP 헤더의 모든 flag를 설정하지 않는 점에서 가져왔다. 즉 ACK , FIN , RST, SYN , URG, PSH 이 설정이 안 되었다는 말이다. 나머지 다른 예약된 비트(RES1, RES2) 는 스캔 결과에 영향을 미치지 않으므로, 설정하는게 별로 중요치 않다. 서버에 null 패킷이 도달하면 BSD 네트워크 코드에서는 열린 포트일 경우 패킷을 버린다.

```
client -> NULL (no flags)
server -> -
```

반대로 닫힌 포트에서는 RST 패킷을 보낸다. 즉 inverse mapping 이다.

```
client -> NULL (no flags)
server -> RST
```

RFC 에서는 이런 종류의 패킷에 대답하는 걸 정의하지 않았기 때문에 운영체제마다 패킷 반응 방법이 다르다.

장점: IDS 를 피할 수 있으며, TCP three-way handshake 를 피할 수 있음
단점: 유닉스만 가능하며, false positive 가 있을 수 있음

Null scan simple code

scan_engine.c

```
void super_scan(struct hoststruct *target, u16 *portarray, int numports, stype scantype)

rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)

broadcast_socket(rawsd);

pd = my_pcap_open_live(target->device, 92,  (o.spoofsource)? 1 : 0, 10);

flt_srchost = target->host.s_addr;
flt_dsthost = target->source_ip.s_addr;
flt_baseport = o.magic_port;
```

```
// 필터 설정
snprintf(filter, sizeof(filter), "(icmp and dst host %s) or (tcp and src host %s and dst host %s and ( dst port %d or dst port %d))", inet_ntoa(target->source_ip), p, inet_ntoa(target->source_ip), o.magic_port , o.magic_port + 1);

// 특정 시각 안에 반응이 오지 않으면 나머지 포트를 열린 포트로 설정
if ( TIMEVAL_SUBTRACT(now, current->sent[current->trynum]) > target->to.timeout)
        current->state = PORT_OPEN;

scanflags = 0;

send_tcp_raw_decoys(rawsd, &target->host, i,
                  current->portno, 0, 0, scanflags, 0, NULL, 0,
                  o.extra_payload, o.extra_payload_length);

while (!timedout && numqueries_outstanding > 0 && ( ip = (struct ip*) readip_pcap(pd, &bytes, target->to.timeout)))

if (tcp->th_flags & TH_RST) newstate = PORT_CLOSED;

// Super_scan 을 사용할 경우 반응이 오지 않으면 open 으로 여기는데
// 이때 방화벽 설정이 엄격한 시스템의 경우 다 열려있는 것으로 나온다.
// 그래서 25개 이상의 Port 가 열린 경우는 엄격한 방화벽이 있는 것으로 확인하고
// 모든 포트를 PORT_FIREWALLED 로 한다.

if (numports > 25) {
// 포트 상태를 PORT_FIREWALLED 로 변환
}
```

XMAS scan

XMAS 스캔은 NULL 스캔의 반대이다. TCP 의 모든 상태 필드(ACK, FIN, RST, SYN, URG, PSH)를 설정하기 때문에 XMAS 혹은 크리스마스 트리 스캔 방식이라 부른다. 예약된 나머지 비트들은 스캔 결과에 별 영향을 주지 않으므로 설정하지 않는다. 이 방법은 BSD 네트워크 코드에 기반하므로 UNIX 호스트에서만 동작한다.

XMAS 스캔은 모든 flag를 설정하고 패킷을 원격 호스트로 보낸다. 만약 상대 호스트의 포트가 열려 있다면 커널은 그 패킷을 버린다. 닫힌 포트로 신호를 보낸 다면, RST 패킷을 받을 수 있다. 이 역시 inverse mapping 방법이다. 그래서 false positive를 잘 구별해야 한다.

```
client -> XMAS (all flags)
server -> -
```

열린 포트일 경우이다. 또는 패킷이 방화벽이나 라우터에서 걸러질 경우이다. 닫힌 포트에서는 다음처럼 응답을 한다.

```
client -> XMAS (all flags)
server -> RST
```

RST 는 클라이언트가 이런 연결에 관한 정보가 없었기 때문에 보내는 것이다. 이 패킷을 받은 즉시 커널에서는 RST 패킷을 클라이언트로 보낸다.

장점: IDS 와 Three way handshake 를 피할 수 있음
단점: UNIX 만 되며, false positive 가 많음

XMAS scan simple code

scan_engine.c

```
void super_scan(struct hoststruct *target, u16 *portarray, int numports, stype scantype)

rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)

broadcast_socket(rawsd);

pd = my_pcap_open_live(target->device, 92,  (o.spoofsource)? 1 : 0, 10);

flt_srchost = target->host.s_addr;
flt_dsthost = target->source_ip.s_addr;
flt_baseport = o.magic_port;

// 필터 설정
snprintf(filter, sizeof(filter), "(icmp and dst host %s) or (tcp and src host %s and dst host %s and
                                                                  ( dst port %d or dst port %d))",
inet_ntoa(target->source_ip), p,
                                                                  inet_ntoa(target->source_ip),
o.magic_port , o.magic_port + 1);

// 특정 시각 안에 반응이 오지 않으면 나머지 포트를 열린 포트로 설정
if ( TIMEVAL_SUBTRACT(now, current->sent[current->trynum]) > target->to.timeout)
        current->state = PORT_OPEN;

// 여러 flag 설정
scanflags = TH_FIN|TH_URG|TH_PUSH;
```

```
send_tcp_raw_decoys(rawsd, &target->host, i,
                   current->portno, 0, 0, scanflags, 0, NULL, 0,
                   o.extra_payload, o.extra_payload_length);

while (!timedout && numqueries_outstanding > 0 && ( ip = (struct ip*) readip_pcap(pd, &bytes, target->to.timeout)))

if (tcp->th_flags & TH_RST) newstate = PORT_CLOSED;

// Super_scan 을 사용할 경우 반응이 오지 않으면 open 으로 여기는데
// 이때 방화벽 설정이 엄격한 시스템의 경우 다 열려있는 것으로 나온다.
// 그래서 25개 이상의 Port 가 열린 경우는 엄격한 방화벽이 있는 것으로 확인하고
// 모든 포트를 PORT_FIREWALLED 로 한다.

if (numports > 25) {
// 포트 상태를 PORT_FIREWALLED 로 변환
}
```

TCP Fragmenting

TCP fragmenting 은 엄밀히 말해서 스캔 방법이 아니다. TCP header를 잘게 쪼개서 스캔하는 과정을 모르게 하는 방법일 뿐이다. 조각난 IP 헤더를 모아서 합하는 과정은 서버에서 예측하기 힘든 비정상적인 결과를 얻을 수 있다. (데이터를 가지고 가는 IP 헤더도 쪼갤 수 있다.) 많은 호스트들이 이런 작은 패킷을 검사하고 다시 모을 수 없다. 경우에 따라서 이런 패킷을 받으면, 시스템이 멈추거나 리붓되기도 한다. 심지어 네트워크 모니터가 다운 되기도 한다. 게다가 이런 작은 패킷은 커널 안의 IP fragmentation 큐에서 근본적으로 막기도 하거나 방화벽 룰셋에서 걸르기도 한다.

많은 침입탐지 시스템에서 스캔 공격 시도를 IP 와 TCP 헤더에 기반을 두고 있으므로, fragmentation 은 이런 방식의 패킷 필터링과 감시를 무력화 시킨다. 그래서 스캔하는 것이 드러나지 않는다.

최소한 작게 작동될 수 있는 TCP 헤더는 첫번째 패킷에 출발 포트와 목적 포트를 포함해야 한다. 즉 (8 octect, 64 bit) 이다. 다른 flag는 나중에 와도 된다. 원격 호스트에서는 도착한 패킷을 조합한다. 실제 패킷을 조합하는 건 IPM (internet protocol module)으로 필드들이 아래의 알맞은 값을 가졌는지 검사한다.

* source
* destination
* protocol
* identification

장점: IDS 를 피할 수 있으며, 탐지되기 어려움
단점: 원격 호스트에서 네트워크 문제를 일으킬 수 있음

Fragmentation Simple code

tcpip.c

```
send_small_fragz(int sd, struct in_addr *source, struct in_addr *victim, u32 seq, u16 sport, u16
dport, int flags)
{

struct pseudo_header {
/*for computing TCP checksum, see TCP/IP Illustrated p. 145 */
  u32 s_addy;
  u32 d_addr;
  u8 zer0;
  u8 protocol;
  u16 length;
};

unsigned char packet[sizeof(struct ip) + sizeof(struct tcphdr) + 100];
struct ip *ip = (struct ip *) packet;
struct tcphdr *tcp = (struct tcphdr *) (packet + sizeof(struct ip));
struct pseudo_header *pseudo = (struct pseudo_header *) (packet + sizeof(struct ip) - sizeof(struct
pseudo_header));
unsigned char *frag2 = packet + sizeof(struct ip) + 16;
struct ip *ip2 = (struct ip *) (frag2 - sizeof(struct ip));
static int myttl = 0;
int res;
struct sockaddr_in sock;
int id;

if (!myttl)  myttl = (time(NULL) % 14) + 51;

sethdrinclude(sd);


/*Why do we have to fill out this damn thing? This is a raw packet, after all */
sock.sin_family = AF_INET;
sock.sin_port = htons(dport);

sock.sin_addr.s_addr = victim->s_addr;

bzero((char *)packet, sizeof(struct ip) + sizeof(struct tcphdr));

pseudo->s_addy = source->s_addr;
pseudo->d_addr = victim->s_addr;
pseudo->protocol = IPPROTO_TCP;
pseudo->length = htons(sizeof(struct tcphdr));

tcp->th_sport = htons(sport);
tcp->th_dport = htons(dport);
tcp->th_seq = (seq)? htonl(seq) : get_random_uint();

tcp->th_off = 5 /*words*/;
tcp->th_flags = flags;

tcp->th_win = htons(2048); /* Who cares */

tcp->th_sum = in_cksum((unsigned short *)pseudo,
             sizeof(struct tcphdr) + sizeof(struct pseudo_header));

// 첫번째 헤더
bzero((char *) packet, sizeof(struct ip));
ip->ip_v = 4;
ip->ip_hl = 5;
ip->ip_len = BSDFIX(sizeof(struct ip) + 16);
```

```
id = ip->ip_id = get_random_uint();
ip->ip_off = BSDFIX(MORE_FRAGMENTS);
ip->ip_ttl = myttl;
ip->ip_p = IPPROTO_TCP;
ip->ip_src.s_addr = source->s_addr;
ip->ip_dst.s_addr = victim->s_addr;

sendto(sd, (const char *) packet,sizeof(struct ip) + 16 , 0, (struct sockaddr *)&sock, sizeof(struct
sockaddr_in)

// 두번째 헤더
bzero((char *) ip2, sizeof(struct ip));
ip2->ip_v= 4;
ip2->ip_hl = 5;
ip2->ip_len = BSDFIX(sizeof(struct ip) + 4); /* the rest of our TCP packet */
ip2->ip_id = id;
ip2->ip_off = BSDFIX(2);
ip2->ip_ttl = myttl;
ip2->ip_p = IPPROTO_TCP;
ip2->ip_src.s_addr = source->s_addr;
ip2->ip_dst.s_addr = victim->s_addr;

sendto(sd, (const char *)ip2,sizeof(struct ip) + 4 , 0, (struct sockaddr *)&sock, (int)
sizeof(struct sockaddr_in
```

5

sweep

스캔을 하기 전에 가장 중요한 것은 그 호스트가 살아있는지 검사하는 것이다. 예전의 경우에는 방화벽이나 IDS 개념이 없었을 경우에는 별 문제가 없었지만, 요즘 처럼 방화벽과 IDS 가 발달한 시기에는 많은 경우 막아 놓고 Filtering 하기 때문에 이런 검사가 필요했고, 방법이 발달했다.

ICMP ping

ICMP echo request 기능을 사용한 방법이다. 작동하고 있는 호스트와 서브넷의 브로드캐스트 주소를 알아 낼 수 있다. ICMP tyep 8 (echo request)의 신호를 보내면 돌아오는 tyep 이 0, 14 , 18 의 경우는 호스트가 살아 있고, 3 이나 11인 경우는 Filtering 되거나 호스트가 네트워크에 연결이 안 된 경우이다.

ICMP ping simple code

targets.c

```
sendpingquery(int sd, int rawsd, struct hoststruct *target,
        int seq, unsigned short id, struct scanstats *ss,
        struct timeval *time, int pingtype, struct pingtech ptech)

icmplen = 8;
pingpkt.type = 8;

pingpkt.code = 0;
pingpkt.id = id;
pingpkt.seq = seq;
pingpkt.checksum = 0;
pingpkt.checksum = in_cksum((unsigned short *)ping, icmplen);

bzero((char *)&sock, sizeof(struct sockaddr_in));
sock.sin_family= AF_INET;
sock.sin_addr = target->host;

send_ip_raw( rawsd, &o.decoys[decoy], &(target->host), IPPROTO_ICMP, ping, icmplen);

get_ping_results(int sd, pcap_t *pd, struct hoststruct *hostbatch, int pingtype, struct timeval
*time, struc
t pingtune *pt, struct timeout_info *to, int id, struct pingtech *ptech, struct scan_lists *ports)

                              if (pd) {
```

```
        ip = (struct ip *) readip_pcap(pd, &bytes, to->timeout);
    } else {
        FD_SET(sd, &fd_r);
        FD_SET(sd, &fd_x);
        res = select(sd+1, &fd_r, NULL, &fd_x, &tmpto);
        if (res == 0) break;
        bytes = read(sd,&response,sizeof(response));
        ip = (struct ip *) &(response);
    }

    if (ip->ip_p == IPPROTO_ICMP) {
        ping = (struct ppkt *) ((ip->ip_hl * 4) + (char *) ip);
        if  ( (ping->type == 0 || ping->type == 14 || ping->type == 18)
                && !ping->code && ping->id == id) {
            if (hostbatch[hostnum].host.s_addr == ip->ip_src.s_addr) {
                pingstyle = pingstyle_icmp;
                newstate = HOST_UP;
            }
         if (ping->type == 3) {

                pingstyle = pingstyle_icmp;
                newstate = HOST_DOWN;
                newportstate = PORT_FIREWALLED;
            } else if (ping->type == 11) {
                        pingstyle = pingstyle_icmp;
                newstate = HOST_DOWN;
            }
```

TCP echo

ICMP 패킷을 보내는 대신에 TCP 패킷을 보내서 확인한다. root 유저가 아닌 일반 사용자는 connect 함수를 이용해서 사용한다.

TCP echo simple code

targets.c

```
sendconnecttcpquery(struct hoststruct *hostbatch, struct tcpqueryinfo *tqi,
            struct hoststruct *target, int seq,
            struct timeval *time, struct pingtune *pt,
            struct timeout_info *to, int max_width)

tqi->sockets[seq] =  socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
unblock_socket(tqi->sockets[seq]);
init_socket(tqi->sockets[seq]);

  bzero(&sock, sockaddr_in_len);
  sock.sin_family = AF_INET;
  sock.sin_port = htons(o.tcp_probe_port);
  sock.sin_addr.s_addr = target->host.s_addr;

  res = connect(tqi->sockets[seq],(struct sockaddr *)&sock,sizeof(struct sockaddr));

if ((res != -1 || errno == ECONNREFUSED)) {
   hostupdate(hostbatch, target, HOST_UP, 1, trynum, to,
        &time[seq], pt, tqi, pingstyle_connecttcp);
} else if (errno == ENETUNREACH) {
   hostupdate(hostbatch, target, HOST_DOWN, 1, trynum, to,
         &time[seq], pt, tqi, pingstyle_connecttcp);
```

```
}

get_connecttcpscan_results(struct tcpqueryinfo *tqi,
                struct hoststruct *hostbatch,
                struct timeval *time, struct pingtune *pt,
                struct timeout_info *to)

                   res = select(tqi->maxsd + 1, &myfds_r, &myfds_w, &myfds_x, &myto);

                   if   (FD_ISSET(tqi->sockets[seq],   &myfds_r)   ||   FD_ISSET(tqi->sockets[seq],
&myfds_w) ||  FD_ISSET(tqi->socket
s[seq], &myfds_x)) {
       res2 = read(tqi->sockets[seq], buf, sizeof(buf));
       if (res2 == -1) {
         switch(errno) {
         case ECONNREFUSED:
         case EAGAIN:
       foundsomething = 1;
       newstate = HOST_UP;
       break;
         case ENETDOWN:
         case ENETUNREACH:
         case ENETRESET:
         case ECONNABORTED:
         case ETIMEDOUT:
         case EHOSTDOWN:
         case EHOSTUNREACH:
       newstate = HOST_DOWN;
       break;
         default:
       snprintf     (buf,     sizeof(buf),     "Strange     read     error     from     %s",
inet_ntoa(hostbatch[hostindex].host));
       perror(buf);
       break;
         }
       } else {
       newstate = HOST_UP;
         }
}
```

TCP SYN

호스트에 SYN 패킷을 보내서 RST 반응이 오는 걸 보고 확인하는 방법이다.

TCP SYN simple code

targets.c

```
sendrawtcppingquery(int rawsd, struct hoststruct *target, int pingtype,
          int seq, struct timeval *time, struct pingtune *pt)

myack = get_random_uint();
myseq = (get_random_uint() << 19) + (seq << 3) + 3;

send_tcp_raw_decoys( rawsd, &(target->host), sportbase + trynum, o.tcp_probe_port, myseq, myack,
TH_SYN, 0, NULL, 0, o.extra_payload, o.extra_payload_length);

get_ping_results(int sd, pcap_t *pd, struct hoststruct *hostbatch, int pingtype, struct timeval
*time,  struct pingtune *pt, struct timeout_info *to, int id, struct pingtech *ptech, struct
scan_lists *ports)
```

```
int newstate = HOST_DOWN;

   if (pd) {
      ip = (struct ip *) readip_pcap(pd, &bytes, to->timeout);
   } else {
      FD_SET(sd, &fd_r);
      FD_SET(sd, &fd_x);
      res = select(sd+1, &fd_r, NULL, &fd_x, &tmpto);
      if (res == 0) break;
      bytes = read(sd,&response,sizeof(response));
      ip = (struct ip *) &(response);
   }

         if (ip->ip_p == IPPROTO_TCP)
   {
                                 newstate = HOST_UP;

   }
```

TCP ACK

포트에 ACK 신호를 보내서 RST 가 오면 그 호스트는 살아있는 걸 의미한다.

TCP ACK simple code

targets.c

```
sendrawtcppingquery(int rawsd, struct hoststruct *target, int pingtype,
          int seq, struct timeval *time, struct pingtune *pt)

myack = get_random_uint();
myseq = (get_random_uint() << 19) + (seq << 3) + 3;

send_tcp_raw_decoys( rawsd, &(target->host), sportbase + trynum, o.tcp_probe_port, myseq, myack,
TH_ACK, 0, NULL, 0, o.extra_payload,
          o.extra_payload_length);

get_ping_results(int sd, pcap_t *pd, struct hoststruct *hostbatch, int pingtype, struct timeval
*time, struc
t pingtune *pt, struct timeout_info *to, int id, struct pingtech *ptech, struct scan_lists *ports)

int newstate = HOST_DOWN;

   if (pd) {
      ip = (struct ip *) readip_pcap(pd, &bytes, to->timeout);
   } else {
      FD_SET(sd, &fd_r);
      FD_SET(sd, &fd_x);
      res = select(sd+1, &fd_r, NULL, &fd_x, &tmpto);
      if (res == 0) break;
      bytes = read(sd,&response,sizeof(response));
      ip = (struct ip *) &(response);
   }

         if (ip->ip_p == IPPROTO_TCP)
   {
                                 newstate = HOST_UP;
   }
```

6
Miscellaneous

이 부분은 open/half-open/stealth 에 포함되지 않는 방법들을 나열한다. 이 방법은 기본부터 다르지만, 오늘날에도 여전히 쓰고 있는 방법이다.

UDP scan

UDP는 포트가 열렸는지 닫혔는지 확인하는 방법이 TCP 와 다르다. UDP는 패킷을 보내는 수단으로 datagram을 사용하므로 연결 지향적인 프로토콜이 아니다. inverse mapping 의 경우와 비슷하게 UDP 패킷을 열린 포트에 보내면 어떤 반응도 얻을 수 없다. 그러나 닫힌 포트에 보내면 ICMP 에러를 얻을 수 있다. 그 과정을 이용해 간단하게 닫힌 포트와 열린 포트를 구별할 수 있다. ICMP 의 메시지 타입은 ICMP_PORT_UNREACH (type 3 code 3) 이다. 그래서 닫힌 포트에 UDP를 보낸 다음 UDP를 기다릴 필요가 없다. UDP 는 연결 도중 패킷이 사라지기 쉽기에 신뢰성이 떨어지는 프로토콜로 알려져 있다. 따라서 재전송이 필요한데, 재전송을 하지 않으면 스캔 결과에서 많은 false positive를 얻을 것이다.

Fyodor에 따르면, 리눅스 커널의 경우는 ICMP 에러 메시지 비율을 제한한다. destination unreachable 이 4초에 80 번 일어 날 경우 그것을 넘을 경우 응답하는데 1/4 의 시간동안 더 지연된다. 열린 포트는 어떤 반응도 보이지 않다. 따라서 재전송하는 패킷으로 false positive를 줄여야 한다.

```
client -> udp packet
server -> -
```

닫힌 포트의 경우는 ICMP 에러로 반응한다.

```
client -> udp packet
server -> ICMP (ICMP_PORT_UNREACH)
```

장점: TCP 포트를 스캔하지 않기 때문에, TCP IDS 를 피할 수 있음.
단점: root 권한이 필요하면 패킷이 쉽게 사라지고 , 쉽게 탐지 될 수 있음.

udp scan simple code

scan_engine.c

```
super_scan(struct hoststruct *target, u16 *portarray, int numports, stype scantype)

rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)

pd = my_pcap_open_live(target->device, 92,  (o.spoofsource)? 1 : 0, 10);

flt_srchost = target->host.s_addr;
flt_dsthost = target->source_ip.s_addr;
flt_baseport = o.magic_port;

snprintf(filter, sizeof(filter), "(icmp and dst host %s) or (tcp and src host %s and dst host %s and ( dst port %d or dst port %d))", inet_ntoa(target->source_ip), p, inet_ntoa(target->source_ip), o.magic_port , o.magic_po
rt + 1);

set_pcap_filter(target, pd, flt_icmptcp_2port, filter);

send_udp_raw_decoys(rawsd, &target->host, i,
                  current->portno, o.extra_payload, o.extra_payload_length);

ip = (struct ip*) readip_pcap(pd, &bytes, target->to.timeout))

icmp = (struct icmp *) ((char *)ip + 4 * ip->ip_hl);

switch(icmp->icmp_code) {

case 3: /* p0rt unreachable */
          if (scantype == UDP_SCAN) {
            newstate = PORT_CLOSED;
          } else newstate = PORT_FIREWALLED;
          break;
}

openlist = testinglist;


for (current = openlist; current;  current = (current->next >= 0)? &scan[current->next] : NULL) {
     addport(&target->ports, current->portno, IPPROTO_UDP, NULL, PORT_OPEN);
}

// Super_scan 을 사용할 경우 반응이 오지 않으면 open 으로 여기는데
// 이때 방화벽 설정이 엄격한 시스템의 경우 다 열려있는 것으로 나온다.
// 그래서 25개 이상의 Port 가 열린 경우는 엄격한 방화벽이 있는 것으로 확인하고
// 모든 포트를 PORT_FIREWALLED 로 한다.

if (numports > 25) {
// 포트 상태를 PORT_FIREWALLED 로 변환
}
```

FTP server bounce scan

이제는 거의 안 쓰이는 이 방법은 hobbit가 제안했다. RFC 959 의 ftp port 명령의 경우 client 가 passive 모드로 설정하면 IP 와 포트 값을 원하는 다른 값으로 변경해서 그 호스트의 포트가 열렸는지 닫혔는지 확인할 수 있다. 만약 데이터 전송의 과정으로 연결이 되었다면 포트는 열린 것이다. 이 경우

150과 226 신호로 응답한다. 연결이 거부되어 실패할 경우에는 425 에러를 낸다.
옛날 버전의 wu-ftp 는 이런 공격에 취약했지만 요즘의 대부분의 ftp 데몬의 경우 패치가 되어서 이런 문제가 없다. 문제가 있는 버전을 나열해 보면

Sun FTP server in SunOS 4.1.x/5.x, SCO OpenServer 5.0.4, SCO UnixWare 2.1, AIX 3.2/4.2/4.2./4.3, Caldera 1.2, RedHat 4.X, Slackware 3.1 - 3.3.

이런 공격을 막는 쉬운 방법은 PORT 명령이 20번 이외의 포트를 못 하게 하면 쉽게 막을 수 있다.

장점: 방화벽을 우회 할 수 있으며 로컬 네트웍을 검사할 수 있고, 추적하기 힘듬.
단점: 느리며 대부분 FTP 데몬에서 패치.

FTP bounce scan simple code

scan_engine.c

```
bounce_scan(struct hoststruct *target, u16 *portarray, int numports, struct ftpinfo *ftp)

snprintf(targetstr, 20, "%d,%d,%d,%d,", UC(t[0]), UC(t[1]), UC(t[2]), UC(t[3]));

for(i=0; portarray[i]; i++) {

portno = htons(portarray[i]);
p1 = ((unsigned char *) &portno)[0];
p2 = ((unsigned char *) &portno)[1];
snprintf(command, 512, "PORT %s%i,%i\r\n", targetstr, p1,p2);
send(sd, command, strlen(command), 0

res = recvtime(sd, recvbuf, 2048,15);
if (recvbuf[0] == '5') {
else  /* Not an error message */
if (send(sd, "LIST\r\n", 6, 0) > 0 ) {
res = recvtime(sd, recvbuf, 2048,12);
if (!strncmp(recvbuf, "500", 3)) {
res = recvtime(sd, recvbuf, 2048,10);
}
if (recvbuf[0] == '1' || recvbuf[0] == '2') {
addport(&target->ports, portarray[i], IPPROTO_TCP, NULL, PORT_OPEN);
if (recvbuf[0] == '1') {
res = recvtime(sd, recvbuf, 2048,5);
recvbuf[res] = '\0';
if (res > 0) {
} else {
addport(&target->ports, portarray[i], IPPROTO_TCP, NULL, PORT_CLOSED);
```

7

Finger Print

원격으로 상대 운영체제를 아는 방법은 여러 가지가 있다. telnet 이나 ftp로 접속하면 나오는 간단한 방법도 있지만, 관리자가 약간 수고를 한다면 다 바꾸고 속일 수 있는 내용들이다. 그래서 원격의 운영체제를 확인하기 위해서 TCP / IP Finger Printing 이라는 기술을 사용한다.

FingerPrinting 방법

FIN 조사

FIN 패킷을(ACK이나 SYN 플래그 없는 어떠한 패킷도 좋다) 열려있는 확인할 포트로 보낸다. RFC793에서는 작동은 응답을 안 하도록 정해졌지만, MS Windows, BSDI, CISCO, HP/UX, MVS, IRIX와 같은 많은 구현에서 RST 패킷을 되돌려 보낸다. 대부분 현재 스캔 도구들은 이 기술을 이용한다.

BOGUS flag 조사

이 방법은 SYN 패킷으로 된 TCP 헤더 안에 정의되지 않은 TCP 플래그(64 또는 128)를 설정하는 것이다. 2.0.35 이전의 리눅스 박스들은 응답 할 때 여전히 이 플래그 설정을 유지한다. 다른 OS에서 이런 버그를 찾아볼 수 없다. 그러나, 몇몇 운영체제에서는 그들이 SYN+BOGUS 패킷을 받았을 때 접속이 끊어진다. 이러한 방법은 그들을 확인할 때 유용하게 쓰일 수 있다.

TCP ISN Sampling

이 방법은 ISN(Initail sequence number) 패턴을 찾는 방법이다. 이것은 (많은 구형의 UNIX박스의 경우) 64000이 증가되는 부류, 랜덤한 증가(Solaris, IRIX, FreeBSD, Digital UNIX, Cray등의 새로운 버전)를 갖는 부류와 진정한 "불규칙"인 (linux 2.0, Open VMS, newer AIX, etc) 부류로 나뉜다. 시간에 의존하는 윈도우 박스(그 외 몇 몇 종류) 모델은 정해진 시간동안 ISN의 증가량이 고정되어 있다. 계산된 변수와 최대 공약수 숫자들과 순차적인 숫자를 이용한 함수를 이용해서 ISN의 랜덤한 증가를 가지는 운영체제의 종류를 더 자세하게 나눌 수 있다.

Don't Fragment bit

많은 운영체제들은 패킷을 보내기 시작할 때 자신들이 보내는 패킷에 "조각 내지 말기" 비트를 설정

한다. 이건 여러 가지 효율 면에서의 이득이 된다. 어떤 경우는 모든 운영체제들이 이것이 적용되지는 않고, 다른 방식으로 하거나, 상대편 운영체제가 어떻게 반응하는 지 본 다음 이 비트를 설정한다.

TCP 시작 윈도우

단순히 되돌아오는 패킷들의 윈도우 크기와 관련된 방법이다. 이 값은 운영체제 종류에 따라 일정하다. 이 검사는 실제로 많은 정보를 주며, 몇몇 운영체제는 윈도우 크기만으로 알아낼 수 있다. (예를 들어 AIX는 유일하게 0X3F25를 사용한다.) 새롭게 TCP 스택을 고쳐 썼다고 말하는 NT 5는 0x402E를 사용한다. 흥미롭게도, 이것은 openBSD와 FreeBSD도 이 값을 쓴다.

ACK 값

완전하게 표준일거라 생각해도, 어떤 경우에 ACK 필드를 사용하는 지에 따라서 다르다. 그 예로, 닫힌 TCP포트에 FIN|PSH|URG를 보내도록 해보면, 윈도우와 몇 몇 프린터는 seq+1을 보내고, 대부분의 다른 운영체제의 경우엔 보낸 ISN과 같은 ISN 값에 ACK Flag를 설정한다. 만약 열린 포트에 SYN|FIN|URG|PSH를 보낸다면 윈도우는 맘대로 작동한다. 때때로 seq로 돌아오거나 다른 때는 seq+1돌아오거나, 다른 때는 전혀 다른 값으로 되돌아온다.

ICMP 에러 메시지 보내기

몇 몇 좋은 운영 체제는 다양한 에러 메시지를 보내는 비율을 RFC 1812에 따라 제한한다. (공격 방어책이라고도 볼 수 있다.) 예로, 리눅스 커널(net/ipv4/icmp.h)는 도달 할 수 없는 목적지라는 에러 메시지를 4초마다 80개로 제한하고 이게 초과하면 1/4 초 더 반응이 느려진다. 이 검사의 한가지 방법은 몇 개의 임의의 높은 UDP포트에 한 묶음의 패킷 보낸 후 되돌아 온 에러 메시지 수를 센다. 이 검사는 운영체제 탐지를 위해 패킷을 한 묶음 보내고 되돌아오길 기다린다. 그렇지만, 네트웍에서 패킷이 사라지는 가능성도 검사해야 한다.

ICMP 메시지 인용

RFC는 여러 다양한 에러로 생기는 몇 가지 ICMP 메시지를 인용하는 ICMP 에러 메시지에 대해 적어 놓았다. ICMP Unreachable 패킷에 대해서, 거의 대부분 운영체제는 오직 IP헤더에 8바이트만 더해서 보낸다. 그러나 솔라리스는 한 비트 더 보내고, 리눅스는 더 많이 붙여 보낸다. 이 흥미 있는 점을 이용해 nmap은 열린 포트가 없어도 위의 두 운영체제를 판별한다.

ICMP 에러 메시지를 되돌려주는 무결성

comp.security.unix에 게재된 Theo De Raadt의 글을 보면. 앞서 말한 것처럼 운영체제는 ICMP unreachable 패킷에 메시지 덧붙여 되돌려 보낸다. 어떤 운영체제는 작동하는 동안에 상대방의 헤더부분을 기반으로 사용하는 경향이 있어서 그들을 되돌려 주면서 약간의 차이가 있게 된다. 예를 들면 AIX와 BSDI는 IP 전체 길이를 20 바이트 더 높게 해서 돌려보낸다. 어떤 BSDI, FreeBSD, OpenBSD, ULTRIX, 와 VAXen 은 IP ID를 엉망으로 만든다. TTL이 변하기 때문에 checksum이 바뀌는 반면에,

어떤 운영체제들(AIX, FreeBSD)은 아무 값이나 돌려보내거나 0 체크 섬을 돌려보낸다. UDP 체크 섬도 마찬가지이다. 백이면 백 NMAP는 이처럼 미묘한 차이를 알아차리는 ICMP에러 상에서 9개의 다른 검사를 한다

서비스 유형(Type of Service)

ICMP unreachable 메시지 부분의 서비스의 유형 부분에서 정보를 얻을 수 있다. 대부분의 모든 운영체제들은 0xc0를 사용하는 리눅스를 빼고, ICMP 에러로 0을 사용한다. 이것은 표준 TOS 값이 이렇다는 것은 아니다. 그러나 사용치 않은 상위필드의 한 부분을 나타낸다. 왜 이것이 설정되어 있는지는 모른다. 그러나 만약 그것들이 0으로 변한다면, 구버전을 확인하도록 유지할 수 있을 것이고 이 방법으로 신(new) 구(old)를 확인할 수 있을 것이다.

조각 관리(Fragmentation Handling)

각기 다른 운영체제들이 종종 IP 조각들을 달리 덮어쓴다는 사실을 이용한다. 어떤 것은 구버전의 부분에 새로운 것을 덮어쓴다. 어떤 경우에는 그 반대일 수 있다. 패킷이 어떻게 다시 조합되는지 알 수 있는 다른 많은 검사들이 있다.

TCP 옵션들(TCP Options)

이 옵션들을 이용해 정보의 쉽게 얻을 수 있는 점에서 아주 좋다.
이 옵션들의 좋은 점은 다음과 같다.

1)모든 운영체제들이 구현되지 않은 일반적으로 선택적인 것이다.
2)운영체제에 옵션에 설정된 대로 질문을 보내서 그것들을 실행한다면 구별할 수 있다. 그 목표 운영체제는 일반적으로 그것의 응답에 설정된 옵션의 지원하는지를 보여준다.
3)한번에 모든 것을 검사 할 수 있도록 하나의 패킷에 모든 옵션을 가득 채울 수 있다.

nmap은 거의 모든 시험 패킷과 함께 이러한 옵션들을 같이 보낸다.

Finger print simple code

nmap 에서는 다음과 같은 파일(nmap-os-fingerprints)과 검사(osscan.c)를 사용한다.

```
*Fingerprint IRIX 6.2 - 6.4 # Thanks to Lamont Granquist
*TSeq(Class=i800)
*T1(DF=N%W=C000|EF2A%ACK=S++%Flags=AS%Ops=MNWNNT)
*T2(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)
*T3(Resp=Y%DF=N%W=C000|EF2A%ACK=O%Flags=A%Ops=NNT)
```

```
*T4(DF=N%W=0%ACK=O%Flags=R%Ops=)
*T5(DF=N%W=0%ACK=S++%Flags=AR%Ops=)
*T6(DF=N%W=0%ACK=O%Flags=R%Ops=)
*T7(DF=N%W=0%ACK=S%Flags=AR%Ops=)
*PU(DF=N%TOS=0%IPLEN=38%RIPTL=148%RID=E%RIPCK=E%UCK=E%*ULEN=134%DAT=E
```

Test 0

첫 번째 줄에는 (관련 부분에 '>'을 표시):

```
> FingerPrint IRIX 6.2 - 6.3 # Thanks to Lamont Granquist
```

간단히 fingerprint가 IRIX 6.2부터 6.3 버전까지 지원한다는 걸 말한다. 그 다음 해석부분은 Lamont Granquist가 IRIX에서 Fingerprints를 검사한 걸 말한다.

```
*> TSeq(Class=i800)
```

ISN(Interal Sequence Number)표본이 i800 class안에 있다는 걸 의미한다. 이것은 각각의 새로운 ISN이 그 전 마지막 ISN보다 800의 배수정도로 커진다는 걸 뜻한다.

Test 0 simple code

```
      while (seq_packets_sent < NUM_SEQ_SAMPLES) {
         if (o.scan_delay) enforce_scan_delay(NULL);
         send_tcp_raw_decoys(rawsd, &target->host,
                        o.magic_port + seq_packets_sent + 1,
                        openport,
                        sequence_base + seq_packets_sent + 1, 0,
                        TH_SYN,
0 ,"\003\003\012\001\002\004\001\011\010\012\077\077\077\077\000\000\00
0\000\000\000" , 20, NULL, 0);
            usleep( MAX(110000, target->to.srtt)); /* Main reason we wait so long is that we need
to spend
more than .5 seconds to detect 2HZ timestamp sequencing -- this also should make ISN sequencing more
regular */

         gettimeofday(&seq_send_times[seq_packets_sent], NULL);
         seq_packets_sent++;

         // 결과를 모은다.
         while(si->responses < seq_packets_sent && !timeout) {

            ip = (struct ip*) readip_pcap(pd, &bytes, oshardtimeout);
            gettimeofday(&t2, NULL);

            lastipid = ip->ip_id;

            if (ip->ip_p == IPPROTO_TCP) {
               tcp = ((struct tcphdr *) (((char *) ip) + 4 * ip->ip_hl));
               if  (ntohs(tcp->th_dport)  <  o.magic_port  ||  ntohs(tcp->th_dport)  -  o.magic_port  >
NUM_SEQ_SA
```

```
MPLES || ntohs(tcp->th_sport) != openport) {
                        continue;
                    }
                    if ((tcp->th_flags & TH_RST)) {

                        } else if ((tcp->th_flags & (TH_SYN|TH_ACK)) == (TH_SYN|TH_ACK)) {

                        seq_response_num = (ntohl(tcp->th_ack) - 2 -
                                            sequence_base);
                        if (seq_response_num < 0 || seq_response_num >= seq_packets_sent) {
                            seq_response_num = si->responses;
                        }
                        si->responses++;
                        si->seqs[seq_response_num] = ntohl(tcp->th_seq); /* TCP ISN */
                        si->ipids[seq_response_num] = ntohs(ip->ip_id);
                        if ((gettcpopt_ts(tcp, &timestamp, NULL) == 0))
                            si->ts_seqclass = TS_SEQ_UNSUPPORTED;
                        else {
                            if (timestamp == 0) {
                                si->ts_seqclass = TS_SEQ_ZERO;
                            }
                        }
                        si->timestamps[seq_response_num] = timestamp;
                        /*              printf("Response  #%d  --  ipid=%hu  ts=%i\n",  seq_response_num,
ntohs(ip->ip_
id), timestamp); */
                        if (si->responses > 1) {
                            seq_diffs[si->responses-2]    =    MOD_DIFF(ntohl(tcp->th_seq),    si->seqs[si-
>responses-2]
);
                        }
                    }
                }
            }
        }

        /* Now we look at TCP Timestamp sequence prediction */
        /* Battle plan:
           1) Compute average increments per second, and variance in incr. per second
           2) If any are 0, set to constant
           3) If variance is high, set to random incr. [ skip for now ]
           4) if ~10/second, set to appropriate thing
           5) Same with ~100/sec
        */

        if (si->ts_seqclass == TS_SEQ_UNKNOWN && si->responses >= 2) {
            avg_ts_hz = 0.0;
            for(i=0; i < si->responses - 1; i++) {
                double hz;

                hz = (double) ts_diffs[i] / (time_usec_diffs[i] / 1000000.0);
                avg_ts_hz += hz / ( si->responses - 1);
            }

            if (avg_ts_hz > 0 && avg_ts_hz < 3.9) { // si-> lastboot 로 켜진 시각도 알 수 있다.
                si->ts_seqclass = TS_SEQ_2HZ;
                si->lastboot = seq_send_times[0].tv_sec - (si->timestamps[0] / 2);
            }
            else if (avg_ts_hz > 85 && avg_ts_hz < 115) {
                si->ts_seqclass = TS_SEQ_100HZ;
                si->lastboot = seq_send_times[0].tv_sec - (si->timestamps[0] / 100);
            }
            else if (avg_ts_hz > 900 && avg_ts_hz < 1100) {
                si->ts_seqclass = TS_SEQ_1000HZ;
                si->lastboot = seq_send_times[0].tv_sec - (si->timestamps[0] / 1000);
```

```
      }
    }

    /* Time to look at the TCP ISN predictability */
    if (si->responses >= 4 && o.scan_delay <= 1000) {
       seq_gcd = gcd_n_uint(si->responses -1, seq_diffs);
       if (seq_gcd == 0) {
          si->seqclass = SEQ_CONSTANT;
       } else if (seq_gcd % 64000 == 0) {
          si->seqclass = SEQ_64K;
       } else if (seq_gcd % 800 == 0) {
          si->seqclass = SEQ_i800;
       } else if (si->seqclass == SEQ_UNKNOWN) {
          seq_avg_inc = (unsigned int) ((0.5) + seq_avg_inc / (si->responses - 1));
          for(i=0; i < si->responses -1; i++)      {
             seq_inc_sum      +=      ((double)(MOD_DIFF(seq_diffs[i],      seq_avg_inc))      *
((double)MOD_DIFF(seq_diffs
[i], seq_avg_inc)));
          }
          seq_inc_sum /= (si->responses - 1);

          /* Some versions of Linux libc seem to have broken pow ... so we
          avoid it */
#ifdef LINUX
          si->index = (unsigned int) (0.5 + sqrt(seq_inc_sum));
#else
          si->index = (unsigned int) (0.5 + pow(seq_inc_sum, 0.5));
#endif

          if (si->index < 75) {
             si->seqclass = SEQ_TD;
             /*   printf("Target is a Micro$oft style time dependant box\n");*/
          }
          else {
             si->seqclass = SEQ_RI;
             /*   printf("Target is a random incremental box\n");*/
          }
       }
```

Test 1

```
*> T1(DF=N%W=C000|EF2A%ACK=S++%Flags=AS%Ops=MNWNNT)
```

이 테스트에서 우리는 SYN패킷과 함께 TCP옵션을 open port로 보낸다. DF=N 은 응답에 "Don't fragment"bit가 설정되어 있지 않다는 걸 의미한다. W=C000|EF2A은 window 반환값이 0xC000 또는 EF2A라야 한다는 걸 의미한다. ACK=S++은 받는 응답 값이 ISN에 1씩 더해진다는 걸 의미한다. Flags = AS 는 ACK나 SYN flags가 응답에 보내진다는 걸 의미한다. Ops = MNWNNT는 응답 속의 옵션이 다음 순서로 된다는 걸 의미한다.

```
<MSS (not echoed)><NOP><Window scale><NOP><NOP><Timestamp>
```

Test 1 simple code

```
/* Test 1 */
if (!FPtests[1]) {
```

```
if (o.scan_delay) enforce_scan_delay(NULL);
send_tcp_raw_decoys(rawsd,&target->host,current_port,openport,sequence_base,
0,TH_BOGUS|TH_SYN,    0,"\003\003\012\001\002\004\001\011\010\012\077\077\077\077
\000\000\000\000\000\000" , 20, NULL, 0);
}
```

Test 2

```
*> T2(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)
```

T1과 같은 상황에서 open port에 NULL로 설정된 패킷을 보낸다. Resp=Y는 응답 값을 얻는다는 걸 말한다. Ops=는 응답 패킷에 어떤 옵션도 포함되지 않는다는 걸 말한다. '%Ops='를 빼면 보낸 어떤 옵션이고 가능하다라는 말이다.

Test 2 simple code

```
/* Test 2 */
if (!FPtests[2]) {
if (o.scan_delay) enforce_scan_delay(NULL);
send_tcp_raw_decoys(rawsd, &target->host, current_port +1,
openport, sequence_base, 0,0, 0,"\003\003\012\001\002\004\001\011\010\012\0
77\077\077\077\000\000\000\000\000\000" , 20, NULL, 0);
        }
```

Test 3

```
*> T3(Resp=Y%DF=N%W=400%ACK=S++%Flags=AS%Ops=M)
```

Test 3은 하나의 open port에 SYN|FIN|URG|PSH window와 options을 보내는 걸 의미한다.

Test 3 simple code

```
/* Test 3 */
if (!FPtests[3]) {
if (o.scan_delay) enforce_scan_delay(NULL);
send_tcp_raw_decoys(rawsd, &target->host, current_port +2,
openport, sequence_base, 0,TH_SYN|TH_FIN|TH_URG|TH_PUSH, 0,"\003\003\012\00
1\002\004\001\011\010\012\077\077\077\077\000\000\000\000\000\000" , 20, NULL,
0);
}
```

Test 4

```
*> T4(DF=N%W=0%ACK=O%Flags=R%Ops=)
```

이것은 open port에 일 경우 커널에서 자동 응답을 한다. 여기엔 Resp=가 없다. 이것은 네크워크 또는 나쁜 방화벽 때문에 생긴 버려진 패킷 같은 응답의 부족 일 경우에도 다른 검사가 끝나기 전까지 판단을 미룬다. 어떤 컴퓨터도 응답을 한다고 생각하기 때문에 이런 옵션을 둔다. 응답의 부족은 네트워크 상태의 속성이지 운영체제 자체의 속성은 아니다. 어떤 운영체제들은 응답 없이 그것들을 버리기 때문에 우리는 Test 2 와 3 에 Resp tag를 넣는다.

Test 4 simple code

```
/* Test 4 */
if (!FPtests[4]) {
if (o.scan_delay) enforce_scan_delay(NULL);
send_tcp_raw_decoys(rawsd, &target->host, current_port +3,
openport, sequence_base, 0,TH_ACK, 0,"\003\003\012\001\002\004\001\011\010\
012\077\077\077\077\000\000\000\000\000\000" , 20, NULL, 0);
}
}
```

Test 5-7

```
*> T5(DF=N%W=0%ACK=S++%Flags=AR%Ops=)
*> T6(DF=N%W=0%ACK=O%Flags=R%Ops=)
*> T7(DF=N%W=0%ACK=S%Flags=AR%Ops=)
```

이들 테스트는 닫혀진 port에 각각 SYN, ACK, and FIN|PSH|URG을 할당한다. 항상 같은 옵션이 지정된다.

Test 5-7 simple code

```
/* Test 5 */
if (!FPtests[5]) {
if (o.scan_delay) enforce_scan_delay(NULL);
send_tcp_raw_decoys(rawsd, &target->host, current_port +4,
closedport, sequence_base, 0,TH_SYN, 0,"\003\003\012\001\002\004\001\011\010\01
2\077\077\077\077\000\000\000\000\000\000" , 20, NULL, 0);
}
```

```
/* Test 6 */
if (!FPtests[6]) {
if (o.scan_delay) enforce_scan_delay(NULL);
send_tcp_raw_decoys(rawsd, &target->host, current_port +5,
closedport, sequence_base, 0,TH_ACK, 0,"\003\003\012\001\002\004\001\011\010\012\077\077\077\077\000\000\000\000\000\000" , 20, NULL, 0);
}

/* Test 7 */
if (!FPtests[7]) {
if (o.scan_delay) enforce_scan_delay(NULL);
send_tcp_raw_decoys(rawsd, &target->host, current_port +6,
closedport, sequence_base, 0,TH_FIN|TH_PUSH|TH_URG, 0,"\003\003\012\001\002\004\001\011\010\012\077\077\077\077\000\000\000\000\000\000" , 20, NULL, 0);
}
```

Test 8

```
*>PU(DF=N%TOS=0%IPLEN=38%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%DAT=E)
```

이 길다란 명령은 'port unreachable'라는 메시지 테스트이다. TOS=0은 서비스 필드와 IP값이 0이라는 뜻이다. 다음 두 개의 필드는 IP헤더의 메시지 길이와 상대 운영체제들이 다시 돌려주는 IP헤더 길이의 값을 조사하라는 말이다. RID=E는 초기의 UDP패킷 복사본에서 우리가 다시 되돌려주는 RID값이 우리가 보낸 것과 일치하는가 조사하라는 말이다. RIPCK=E는 checksum(확인값, 반복값)이 변경되지 않는 걸 의미한다.(만약에 변경된다면 RIPCK=F라고 적는다) UCK=E는 UDP checksum이 정확하다는 걸 의미한다. 다음에 오는 0x134이라는 UDP 길이와 DAT=E은 정확하게 우리의 UDP data값을 되돌려 주는 지를 의미한다. 이 경우를 포함한 대부분의 프로그램이 어떤 UDP data도 돌려주지 않기 때문에 기본적으로 DAT=E라 기술한다.

Test 8 simplde code

```
/* Test 8 */
if (!FPtests[8]) {
if (o.scan_delay) enforce_scan_delay(NULL);
upi = send_closedudp_probe(rawsd, &target->host, o.magic_port, closedport);
}
gettimeofday(&t1, NULL);
timeout = 0;
```

8

맺음말

막상 다 쓰고 나니, 너무 부족한 부분이 많다. 소스를 깊게 연구를 했으면 좋았을 텐데라는 아쉬움이 남는다. RPC 부분은 그 내용이 방대하고 따로 공부해야 될 내용이 많아서, 나중 문서에 추가하기로 하였다. IDLE scan 부분도 새로운 내용이라 소스를 분석하는 시간이 모자랐다. 이 역시 나중에 추가할 생각이다.

문서 처음에 얘기했던 이론 부분은 어느 정도 완성이 되었지만, 실제적으로 쓰는 부분이 미흡하다. 간단한 nmap 옵션에 대해서도 적어 놓지를 않아서, nmap 프로그램을 모르고 문서를 접하는 사람에게 이론적으로 도움만 될뿐 실제적인 스캔엔 별로 도움이 되지 않는다. 이 역시 나중에 추가할 생각이다.

문서를 쓰면서, 문서 번역부분과 오탈자를 지적해준 rainbow 군과 가끔씩 도와준 polonaiz 군에게 감사를 드리고, KHDP 와 Nunll@Root 그리고 HSD 멤버들에게도 감사하며, 이만 문서를 마친다.

부족한 내용이나, 궁금한 내용은 oprix@hanmail.net 으로 연락바라며 문서에 "[ ]" 이런 문자가 들어가면 걸러지니 제목을 잘 정해서 보내주기 바란다. 덧붙여서 전지현 사진도 함께 보내 주시면 좋겠다. :')

9

참고 문헌

[1] Phrack (phrack.org)
49−15 TCP Port stealth scanning
51−11 The Art of Scanning
54−9 Remote OS detection via TCP/IP Stack FingerPrinting

[2] Synnergy(synnergy.net)
Examining port scan methods
http://www.synnergy.net/downloads/papers/portscan.txt

[3] Certcc
Nmap 네트워크 점검 도구 및 보안 스캐너
http://www.certcc.or.kr/tools/Nmap.html

[4] Addison−wesley
TCP IP Illustrated Volume 1

[5] O'Reilly – 한빛미디어
Unix Systems Programing for SVR4
David A. Curry. 이수진, 이성희 역

[6] RFC
793 TRANSMISSION CONTROL PROTOCOL
http://www.ietf.org/rfc/rfc793.txt
959 FILE TRANSFER PROTOCOL (FTP)
http://www.ietf.org/rfc/rfc959.txt
1413 Identification Protocol
http://www.ietf.org/rfc/rfc1413.txt